University of Alberta

An Extendible Hashing Structure for Image Similarity Searches

by

Shu LIN

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2000

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-59834-9

Canada

University of Alberta

Library Release Form

Name of Author: Shu LIN

Title of Thesis: An Extendible Hashing Structure for Image Similarity Searches

Degree: Master of Science

Year this Degree Granted: 2000

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Shu LIN
615 General Services Building
Edmonton, Alberta
Canada, T6G 2H1

Date: May 25th, 2000

Abstract

Image similarity search is employed to retrieve similar images from a database. Colour features, particularly colour histograms, are extensively utilized to compute the image similarity. This thesis presents a multi-scale representation of image colour histograms that allows queries at different precision levels as well as queries on part of an image instead of an entire image.

Because image databases usually store a large number of images, sequential scanning of the database is not feasible; an index scheme is needed to speed up the similarity search. A new index structure — three-dimensional extendible hash — is designed to index image average colours. The average colours are used in a two-step colour similarity search: the index is searched first to filter the database before colour histogram comparisons are applied. Experiments have been made to test the performance of the three-dimensional extendible hash, and the results are discussed in this thesis.

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **An Extendible Hashing Structure for Image Similarity Searches** submitted by Shu LIN in partial fulfillment of the requirements for the degree of **Master of Science**

Dr. M. Tamer Özsu

Dr. Mrinal Mandal

Dr. Osmar R. Zaïane

Date: 24 May 2000

Acknowledgements

I would like to express my gratitude to the members of the database group who have helped me complete this thesis.

Dr. Tamer Özsu for being a great supervisor.

Dr. Vincent Oria for being a great "unofficial" supervisor.

Dr. Osmar R. Zaïane and Dr. Mario A. Nascimento for their helpful advice.

Paul Iglinski for his technical support.

Anne Nield for proofreading my thesis.

Bin Yao for companying me in the lab during the tedious programming time.

I also would like to express my appreciation to Guangjun Cao, a Ph.D. student in the Department of Mathematical Sciences, who acted as my mathematical consultant during the research.

Contents

1 Introduction 1

1.1 Motivation . . . 1

1.1.1 Content-based Image Retrieval . . . 1

1.1.2 Similarity Queries . . . 2

1.1.3 Indexing Image Feature Vectors . . . 3

1.2 Thesis Scope . . . 3

1.3 Thesis Contributions . . . 4

1.4 Thesis Organization . . . 4

2 Related Work 6

2.1 Multi-dimensional Indexing Structure . . . 6

2.1.1 R-trees, R^+-trees, and R^*-trees . . . 7

2.1.2 SS-Trees . . . 11

2.1.3 SR-Trees . . . 11

2.1.4 Grid Files . . . 14

2.2 Dimensionality Reduction and Filtering . . . 16

2.3 Indexing in Metric Spaces . . . 19

3 Colour Histogram Comparisons 22

3.1 Colour Spaces . . . 22

3.2 Colour Histograms . . . 24

3.3 Colour Histogram Distance Metrics . . . 26

3.4 Multi-scale Colour Histograms . . . 28

3.5 Multi-precision Similarity Queries . . . 29

3.6 Sub-image Similarity Queries . . . 31

4 Multi-dimensional Extendible Hashing 34

4.1 Three-dimensional Extendible Hashing . . . 34

4.2 Multi-dimensional extendible hashing . . . 41

4.3 Range Query Algorithm . . . 41

4.4 Multi-dimensional Linear Hashing . . . 45

5 Experiments **49**
5.1 Experiment Setup 49
5.2 Experiment Results and Discussion 53
5.2.1 Query Performance 53
5.2.2 Space Utilization 57

6 Colour Histogram Similarity Queries in the DISIMA System **66**
6.1 The DISIMA Model 66
6.1.1 The Model Components 66
6.1.2 The Type System Overview 68
6.2 The DISIMA Architecture 68
6.3 MOQL and VisualMOQL 69
6.4 Integration of Indexing Structure 75

7 Conclusions and Future Work **77**

Bibliography **79**

A **83**

List of Tables

5.1 Disk blocks used by SR-tree and Hash 53

List of Figures

2.1 Data rectangles (adapted from [20]) 8
2.2 An R-tree formed from Figure 2.1 (adapted from [20]) 8
2.3 Data rectangles 10
2.4 An R^+-tree formed from Figure 2.3 10
2.5 Data rectangles (adapted from [40]) 12
2.6 An SS-tree formed from 2.5 ((adapted from [40]) 12
2.7 Intersection of a bounding rectangle and a bounding sphere (from [23]) . . . 13
2.8 An SR-tree formed from 2.7 (from [23]) 13
2.9 Grid file (from [18]) 15
2.10 Buckets in grid file 16
2.11 Illustration of the Karhunen-Loève transformation 17
2.12 Distance ordering not preserved when projecting 18
2.13 Small range in dense portion; large range in sparse portion 19
2.14 An example trie with four objects and two keys (from [3]) 20

3.1 The RGB colour space cube 23
3.2 The HSV colour space: hex-cone 23
3.3 Colour gamut in u^*v^* coordinate system 24
3.4 A three-dimensional representation of $L^*u^*v^*$ colour 24
3.5 Histogram of an image in which warm colours are dominant 25
3.6 Two images with different colour layouts have the same colour histogram . 25
3.7 Colour Histograms whose perceptual similarities are not consistent with their L_1 or L_2 distance 26
3.8 The 4-level multi-scale representation 28
3.9 A quadtree stores colour histograms of image blocks 29
3.10 Three search strategies (adapted from [24]) 32
3.11 Specify portion of interest on the 8-by-8 grid 32

4.1 Hashing average colour into buckets 35
4.2 Initial hash directory of the three-dimensional extendible hashing 36
4.3 Expanded hash directory of the three-dimensional hashing 38
4.4 Expanded hash directory of the three-dimensional hashing 39

4.5 Expanded hash directory of the three-dimensional hashing 40
4.6 Merging disk blocks and shrinking directory 42
4.7 Initial partitions of the colour space 43
4.8 First step of the search algorithm 43
4.9 Further partitioning of initial partitions 44
4.10 Split disk blocks 44
4.11 Non-split disk blocks 45
4.12 The growth of address space in multi-dimensional linear hashing 48

5.1 Initial colour space partition 50
5.2 Average colour distributions 51
5.3 Real data (6601 images) distribution 52
5.4 I/O performance of Hash and SR-tree 54
5.4 I/O performance of Hash and SR-tree (continued) 55
5.4 I/O performance of Hash and SR-tree (continued) 56
5.5 CPU time performance of Hash and SR-tree 58
5.5 CPU time performance of Hash and SR-tree (continued) 59
5.5 CPU time performance of Hash and SR-tree (continued) 60
5.6 Scalability of Hash regarding I/O performance 61
5.7 Scalability of Hash regarding CPU time 62
5.8 Space utilization of a continuously growing dataset 64
5.9 Space utilization of a steady-state dataset 65

6.1 DISIMA model overview 67
6.2 An example of image hierarchy 67
6.3 An example of salient object hierarchy 68
6.4 DISIMA type system overview 69
6.5 The DISIMA architecture 70
6.6 VisualMOQL interface 72
6.7 Query translation 73
6.8 Image property dialog 74

Chapter 1

Introduction

1.1 Motivation

1.1.1 Content-based Image Retrieval

In the past two decades, database systems have proved to have great practical value for handling large amounts of data. Recent advances in image acquisition, storage, processing, and display capabilities have made the use of digital images affordable and widespread, which increases the need to manage the images in database systems.

Traditional databases can deal only with alphanumeric data. A straightforward way of applying traditional database techniques for image retrieval is to retrieve images by keywords or by matching text annotations. The keywords or text annotations are manually associated with images. However, such an approach is fraught with difficulties. First, manually annotating images is a time-consuming task and the annotations are highly subjective. Second, some visual aspects of images are inherently difficult to describe, while others are equally well described in many different ways [29].

Given the huge volume of image data that exist now, and will be collected in the near future, the approach of relying on human-provided annotations is inadequate. Efficient and effective image retrieval should utilize computer-automated extracted image visual features such as colour, texture, shape, and spatial relationship [6]. This is termed as *content-based image retrieval*. However, since the current image processing is not mature enough, we have to somewhat rely on text annotations to provide image content semantics [6]. Thus the current image retrieval method is actually a combination of content-based approaches and keyword-based or other existing approaches.

In recent years, there have been many attempts to build image database systems that support content-based image retrieval—such as Photobook system [33], VisualSEEK system [37], C-BIRD system [43, 26], and DISIMA system [31].

1.1.2 Similarity Queries

In image retrieval context, exact queries, which aim at locating exact copies, are usually less meaningful than similarity queries, which aim at locating images that are "similar" to a given prototype image. This approach is also referred to as *query by example*.

To support similarity queries, similarity metrics between different images are defined. A well-known theory postulates that human similarity perception is based on the measurement of an appropriate distance in a metric psychological space [6]. In this theory, it is assumed that multi-dimensional image feature vectors (points in a multi-dimensional space) are used to model the stimulus' properties. Proximity in the multi-dimensional vector space reflects the similarity of the images; distance in feature vector space measures dissimilarity. Suppose $X(x_1, x_2, \ldots, x_n)$ and $Y(y_1, y_2, \ldots, y_n)$ are two image feature vectors. The commonly used distance functions are formulated as follows

- the city-block distance (L^1 norm): $d(X,Y) = \sum_{i=1}^{n} |x_i - y_i|$;
- the Euclidean distance (L^2 norm): $d(X,Y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$.

Similarity is usually measured by a value between 0 and 1, which is calculated from distances as follows:

$$similarity = 1 - \frac{d}{d_{max}} \tag{1.1}$$

where d is the distance and d_{max} is the maximum of the distances. There may be other ways of defining similarity in terms of distance, as long as similarity is defined as a decreasing function of distance. Although people perceive images from the perspective of similarity, implementation of image similarity queries usually deals with "minimum dissimilarity" rather than "maximum similarity".

There are basically two different types of similarity queries. One is *range queries*: given a query image, search a collection of images to find the ones that are within a user-defined distance from the query image. Formally, given a query image q, a distance threshold t, and an image database D, a range query tries to find the answer set S defined as

$$S = \{i | d(q,i) \leq t, i \in D\}$$

Users may specify similarity threshold instead of distance threshold. In this case, the similarity threshold is translated to distance threshold. If Equation 1.1 is used, similarity threshold is translated to distance threshold by

$$t_d = d_{max} - t_s \times d_{max}$$

where t_d is distance threshold and t_s is similarity threshold.

The other type of similarity query is the k-nearest neighbour query. This type of query tries to retrieve k images which are the most similar to the query image (k-nearest neighbours). Formally, given a query image q and an image database D, a k-nearest neighbour

query tries to find the answer set S which contains k images such that

$$\forall i \in S, \forall j \in D \wedge j \notin S \Longrightarrow d(q,i) \leq d(q,j), \text{ where } \|S\| = k.$$

Usually the images returned by similarity queries will be sorted in order of decreasing similarity.

1.1.3 Indexing Image Feature Vectors

If similarity queries are processed without index structures, each image in the database has to be compared to the query image. For a small size database, this is acceptable. However, as the database grows in size, sequential scanning of the entire database quickly breaks down as an effective search strategy. Index structures must be used to facilitate similarity searches.

Given that images are modeled by feature vectors (multi-dimensional points), image similarity queries can be answered by searching for the close feature vectors in the multi-dimensional space. This approach for content-based retrieval reduces the problem to searching in multi-dimensional space. Index structures are then developed to index image feature vectors, in order to speed searches in multi-dimensional space.

There have been many studies on indexing within the context of traditional relational database management systems. The most popular indexing techniques are based on B-trees [14]. However, these index structures do not apply for image feature vectors. This is because traditional index structures only deal with one-dimensional data, while feature vectors are multi-dimensional. There have also been many index structures developed for spatial data [34]—such as k-d trees, quadtrees, grid files, and R-trees. Although these index structures can handle multi-dimensional data, they may not be efficient under the particular distribution of image feature vectors in the multi-dimensional space. In addition, the dimensionality of image feature vectors is typically very high. The index structures developed for spatial data can not efficiently handle high-dimensional data. So further research on developing index structures to support efficient image similarity queries is necessary.

1.2 Thesis Scope

This thesis concentrates on developing an index scheme to support efficient image retrieval based on colour histograms.

Colours are one of the visual features that people immediately perceive when looking at an image. Colour histograms, which encompass colour distribution information about images (see Section 3.2), are extensively used in image retrieval based on colour similarity. For each image in the database, a colour histogram is automatically computed. These colour histograms are stored in the database along with the images. For the example image, the colour histogram is computed on-the-fly. The (dis)similarity between the example image and the database image is measured by the distance of the corresponding colour histograms.

Colour histograms are high-dimensional data, usually no less than 64 dimensions. The current multi-dimensional index structures, however, are not efficient for high-dimensional data. Indexing directly on colour histograms, therefore, is not feasible. Instead, a filtering approach is employed. Indexing is built on average colours (three-dimensional data) instead of colour histograms. Average colours can be considered as a compact representation of colour histograms. A similarity query based on colour histograms is translated to a coarse-grained similarity query on average colours. Such a query can be processed efficiently with the help of indexing. The resulting intermediate small set of images are then sequentially scanned to compute the final result. A new index structure, *three-dimensional extendible hash*, is designed to index average colours. Experiments are run to examine the performance of the three-dimensional extendible hash and compare it to the SR-tree [23], which is one of the best multi-dimensional index structures to date. The results show that the three-dimensional extendible hash outperforms the SR-tree with respect to CPU time. For I/O performance, three-dimensional extendible hash outperforms the SR-tree under certain circumstances.

It is important to mention that finding the appropriate colour histogram representation, and defining the correct distance metric on the colour histograms, is an area of research in itself. This thesis will discuss the issue of colour histogram comparisons, but does not focus on that concern.

1.3 Thesis Contributions

The main contribution of this thesis is the design of the new three-dimensional hashing structure. This structure is actually be able to handle multi-dimensional data; three-dimension is just a special case. The experiments run on three-dimensional hashing indicate that this structure is promising.

The other contribution of this thesis is proposing two kinds of colour histogram similarity queries: multi-precision similarity queries and sub-image similarity queries. It is also proposed in this thesis that multi-scale colour histograms [24] be used to support these two kinds of queries.

1.4 Thesis Organization

This thesis consists of six chapters followed by a bibliography and appendix. Chapter 1 is the introduction, which describes background knowledge and defines the problem.

Chapter 2 presents the review of related work. It introduces several representative multi-dimensional index structures, dimensionality reduction techniques, and filtering approaches which are used in conjunction with the multi-dimensional index, and another totally different indexing scheme—indexing in metric spaces.

Chapter 3 introduces colour spaces and colour histograms, and discusses colour histogram comparison metrics. It also presents the concept of multi-scale colour histograms and discusses its application to multi-precision similarity queries and sub-image similarity queries.

Chapter 4 presents the new index structure—three-dimensional extendible hashing. It also describes the extension of the structure to deal with multi-dimensional data. Another extension presented applies the crucial structure of three-dimensional extendible hashing to linear hashing. This results in another new index structure—multi-dimensional linear hashing.

In Chapter 5, the three-dimensional extendible hashing is compared to the SR-tree by running experiments on both of them and analyzing the experimental results. The focus of the comparisons is on query performance in terms of both I/O and CPU time. Experiments are also conducted to test the space utilization of three-dimensional extendible hashing.

In Chapter 6, the DISIMA (Distributed Image Database Management) system, which is developed by the Database Research Group at the University of Alberta, is introduced, and the extension of DISIMA system which handles colour histogram similarity queries is described.

Finally, Chapter 7 concludes the thesis and points out possible further enhancements of three-dimensional extendible hashing.

Chapter 2

Related Work

Images are often modeled by their visual features such as colour, texture, and shape. These visual features are represented as points in a multi-dimensional vector space. To support efficient image search, multi-dimensional index structures, known as point access methods (PAMs) [34, 18] are employed to index image feature vectors. The current multi-dimensional indexing structures are generally efficient for low-dimensional data, but their query performance degrades as the dimensionality of data increases. Dimensionality reduction techniques and filtering approaches are used to circumvent this problem. The other way to deal with this difficulty is indexing in metric spaces instead of multi-dimensional vector spaces.

2.1 Multi-dimensional Indexing Structure

Over the past few years, multi-dimensional indexing structures has been extensively studied. The main difficulty encountered in designing multi-dimensional indexing structures is that, unlike the case in one-dimensional space, there exists no total ordering among multi-dimensional points that preserves their proximity. In other words, there is no mapping from two- or higher- dimensional space into one-dimensional space, such that any two objects that are close in the higher-dimensional space are also close to each other in the one-dimensional sorted sequence. Multi-dimensional indexing structures usually employ "bucket methods" [34]. The points in the database are organized into a number of buckets, each of which corresponds to some subspace of the universal multi-dimensional space. The subspaces are often referred to as bucket regions, even though their dimensionality may be greater than two. Through partitioning the universe into bucket regions, the multi-dimensional indexing structures potentially achieves clustering: separating objects which are far apart and grouping objects which are close to each other.

All known access methods appear to fall into one of the two broad categories: hierarchical and non-hierarchical indexing. In the following, we present R-trees[20], R^+-trees[35], R^*-trees[2], SS-Trees[40], and SR-trees[23] as representatives of hierarchical methods, and grid files[30] as a representative of non-hierarchical methods.

2.1.1 R-trees, R^+-trees, and R^*-trees

R-trees [20] were originally proposed to index n-dimensional rectangles. They can be used to index n-dimensional points as well if points are considered as zero size rectangles.

In R-trees, n-dimensional space is partitioned into n-dimensional rectangles that may overlap. An R-tree corresponds to a hierarchy of nested rectangles. The root node corresponds to a rectangle encompassing every other rectangle in the space. Each node in R-trees corresponds to the smallest rectangle that encloses its child nodes. The leaf nodes contain the data objects (or pointers to the data objects) and their minimum bounding rectangles, which are referred to as *data rectangles*. A data rectangle can intersect or be contained in many bounding rectangles, but is stored in only one leaf node. Figure 2.1 illustrates the containment and overlapping relationships between the rectangles, in which data rectangles are shown by a solid line, and dashed rectangles are the bounding rectangles. Figure 2.2 shows an R-tree formed from Figure 2.1. The basic rules for the formation of an R-tree are the following:

- An R-tree is a height-balanced tree; that is, all leaf nodes are at the same level.
- An R-tree order (m, M) where $m \leq \frac{M}{2}$ indicates that each non-leaf node has between m and M children, and each leaf node contains between m and M data objects.
- The root node has a least two child nodes, unless it is a leaf node.

The search algorithm is straightforward. It descends the tree from the root and visits nodes whose corresponding rectangles contain the target data rectangle. More than one subtree under a node may need to be searched, because the data rectangle may be in the overlapping part of bounding rectangles. For example, $R9$ is contained in both $R3$ and $R4$ (Figure 2.1). When searching for $R9$ in the R-tree shown in Figure 2.2, the search algorithm do not know in which subtree $R9$ is stored, so both $R3$ and $R4$ subtrees need to be searched, although $R9$ is in the $R3$ subtree only.

R-tree is a fully dynamic data structure, which means that inserts and deletes can be intermixed with searches, and no periodic reorganization is required. The algorithm for inserting begins by descending the tree to locate a leaf to accommodate the new data rectangle. In contrast to searching, we traverse only a single path from the root to the leaf. At each step of descending the tree, we choose the subtree whose corresponding rectangle needs the least enlargement to include the new data rectangle. If several satisfy this criterion, the rectangle with the smallest area is chosen. Once the leaf node has been determined, a check is made to see if the leaf has room for the new rectangle. If it does, the new data rectangle is added to the leaf node. If this requires an enlargement of the corresponding bounding rectangle, we adjust it appropriately and propagate the change upwards. If the leaf node already has M data rectangles, then the node must be split, and the $M + 1$ rectangles must be distributed to the two nodes. The splitting criterion is that

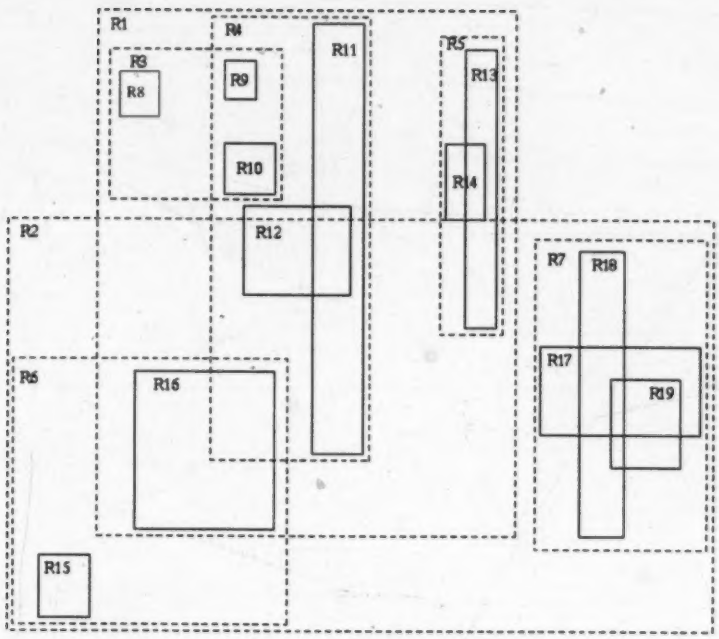


Figure 2.1: Data rectangles (adapted from [20])

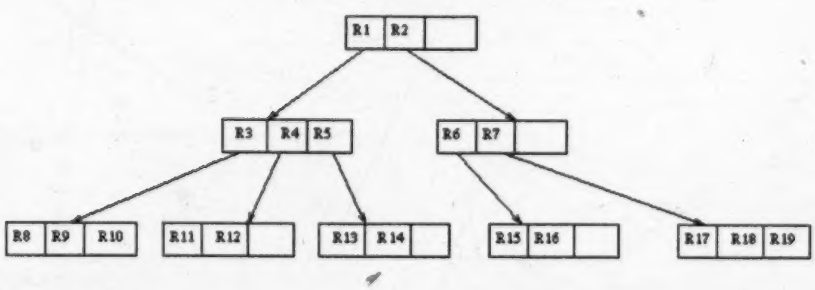


Figure 2.2: An R-tree formed from Figure 2.1 (adapted from [20])

the total area of the two bounding rectangles after a split should be minimized. The split is propagated up the tree. If node split propagation causes the root to split, we create a new root whose child nodes are the two resulting nodes.

Deletion of a data rectangle from an R-tree proceeds by locating the leaf node containing the rectangle to be deleted, and removing this rectangle. If the deletion causes no underflow, we check whether the corresponding bounding rectangle can be reduced in size. If so, we adjust the rectangle and propagate the change up the tree. If underflow occurs, we copy the underflow node into a temporary set and remove the node from the tree. The node removal is propagated up the tree. We then reinsert the rectangles in the temporary set. The data rectangles are added into leaf nodes. The bounding rectangles are placed higher in the tree so that leaves of their dependent subtrees will be on the same level as leaves of the main tree. The reinsertion enables the tree to reflect dynamically the changing spatial structure of the data, rather than the gradual deterioration that might occur if a rectangle was located under the same parent throughout its lifetime.

The problem with R-trees is that a large number of nodes may have to be examined when searching for a data rectangle, since the data rectangle may be contained in the bounding rectangles of many nodes while it is stored in only one of the leaf nodes. To avoid this problem, R^+-trees [35] are proposed. R^+-trees do not allow overlap among bounding rectangles. Each data rectangle is associated with all the bounding rectangles with which it intersects. Thus a data rectangle may appear in several leaf nodes. Figure 2.3 shows the data rectangles whose layout is the same as those in Figure 2.1, as well as the bounding rectangles. Figure 2.4 is the R^+-tree built for Figure 2.3. Notice that rectangle $R11$ is stored in the $R3$, $R4$, and $R6$ subtrees. Unlike R-trees, R^+-trees no longer guarantee the minimum node occupancy and thus are typically higher than R-trees; however R^+-trees speed up the retrieval time.

Based on a study of R-tree behavior under different data distributions, R^*-trees [2] have been proposed to improve R-trees. Like R-trees, R^*-trees also permit overlap among bounding rectangles, but they use a more sophisticated insertion algorithm to enhance performance. The insertion algorithm introduces a policy called "forced reinsert": if a node overflows, instead of splitting it right away, p rectangles which are the farthest away from the center of the bounding rectangle are removed from the overflowing node and reinserted into the tree. The parameter p may vary. The belief behind the idea of forced reinsert is that the structure of R-tree depends on the order of insertions and deletions. Data rectangles inserted during the early growth of the structure may have introduced some bounding rectangles that are not likely to guarantee a good retrieval performance in the current situation. This is also why R-trees employ reinserts in the deletion algorithm. By using reinserts in the insertion algorithm as well as in the deletion algorithm, better dynamic reorganizations of R^*-trees are achieved.

When overflowing happens again, at a level on which the forced reinsert has been applied

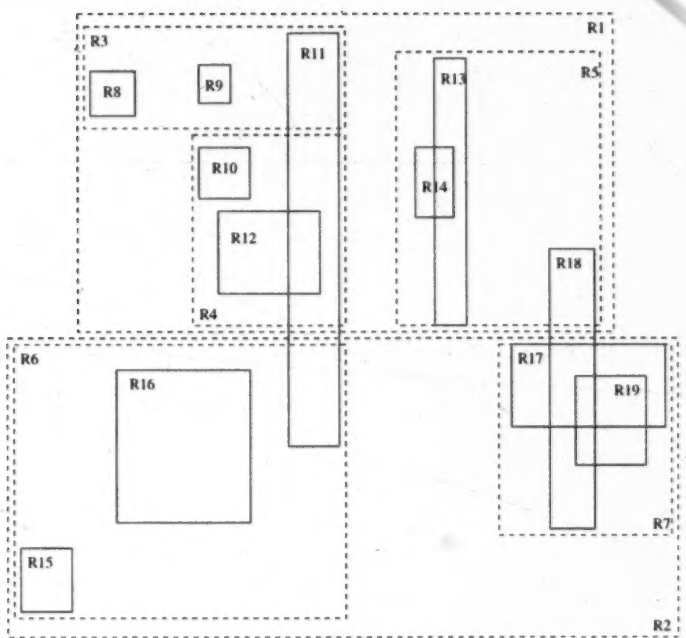


Figure 2.3: Data rectangles

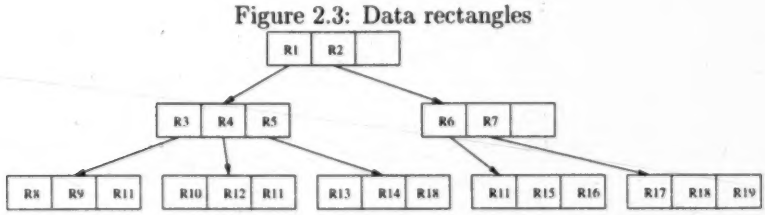


Figure 2.4: An R^+-tree formed from Figure 2.3

during an insertion, the overflowing node is then split. The split policy of R^*-trees is more complex than that of R-trees. Besides minimizing the area covered by the bounding rectangles, R^*-trees adopt the following optimization criteria:

- The overlap between bounding rectangles at the same tree level should be minimized. This decreases the number of search paths to be traversed.
- The perimeter of a bounding rectangle should be minimized. Thus the preferred shape of a rectangle is a square. The minimization of the perimeter will improve the structure: since square-like rectangles can be packed easier, the rectangle that bounds them will have a smaller area.
- Storage utilization should be optimized. Higher storage utilization will generally reduce the query cost, as the height of the tree will be kept low.

Often the tree nodes correspond to disk blocks if the index is disk-resident. The R-tree and its variances are suitable for storing data on disk since they preserve ordering: rectangles close to each other are more likely to be stored together on one disk block.

2.1.2 SS-Trees

Similarity search trees (SS-trees) [40] are specially designed to support similarity query in high-dimensional visual feature space. SS-trees index high-dimensional vectors (points). Their structure is similar to that of R^*-trees except that SS-trees use bounding spheres instead of rectangles to group the points. The center of a sphere is the centroid of the underlying points. Figure 2.5 shows points in two-dimensional space while Figure 2.6 shows a SS-tree created for the points in Figure 2.5.

The search algorithm for SS-tree is very similar to that for R^*-trees. It traverses the tree from the root to the leaves, visiting the nodes which may contain target points. The insertion algorithm is similar to R^*-trees in that it also uses the concept of forced reinsert. The difference is that R^*-trees perform reinsertion unless reinsertion has been made on the same tree level, while SS-trees perform reinsertion unless reinsertion has been made at the same node. SS-trees perform reinsertion more frequently than R^*-trees. This improves the dynamic reorganization of the tree structure. When determining a subtree to accommodate the new point, the algorithm chooses the one whose centroid is the closest to the new point. SS-trees split nodes along the dimension which has the highest variance, and the split location is chosen such that the sum of variances on each side of the split is minimized.

It is reported [40] that SS-trees have better query performance than R^*-trees.

2.1.3 SR-Trees

The SR-tree [23] is a combination of the SS-tree and the R^*-tree in that it uses both bounding spheres and bounding rectangles to group data points. Using spheres to bound data

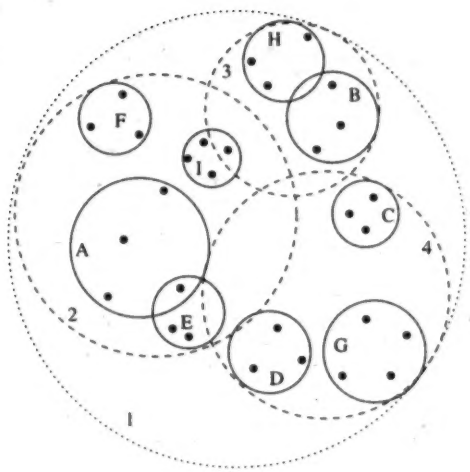


Figure 2.5: Data rectangles (adapted from [40])

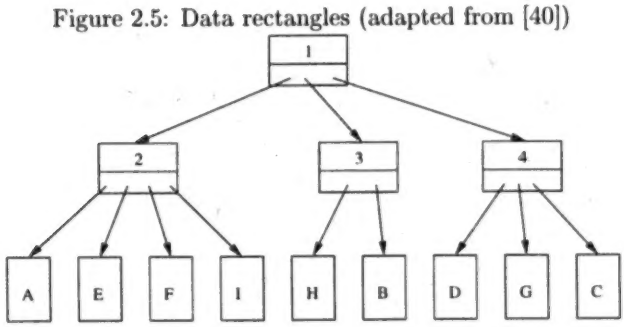


Figure 2.6: An SS-tree formed from 2.5 ((adapted from [40])

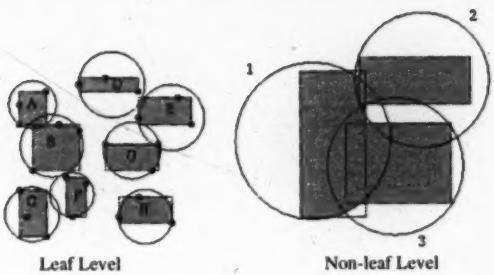


Figure 2.7: Intersection of a bounding rectangle and a bounding sphere (from [23])

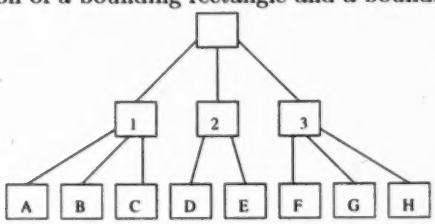


Figure 2.8: An SR-tree formed from 2.7 (from [23])

points is more suitable for similarity searches than using rectangles, since similarity search regions are usually spheres too. However, bounding spheres occupy much a larger volume than bounding rectangles when the dimensionality is high. Regions with larger volume tend to produce more overlap among themselves, which reduces similarity search efficiency. Katayama *et al.* propose the SR-tree [23] to solve this problem by integrating bounding spheres and bounding rectangles. The SR-tree specifies a region by the intersection of a bounding sphere and a bounding rectangle. The introduction of bounding rectangles permits neighborhoods to be partitioned into smaller regions than the SS-tree and improves the disjointness among regions.

The structure of SR-trees is similar to R^*-trees and SS-trees except that each node of SR-trees corresponds to the intersection of a bounding sphere and a bounding rectangle instead of a sphere or a rectangle alone, as shown by Figure 2.7 and Figure 2.8.

A leaf of the SR-tree consists of a number of entries. Each entry contains a point and its attribute data. A non-leaf node of the SR-tree also consists of a number of entries, while each entry corresponds to a child of the node and consists of four components: a bounding sphere, a bounding rectangle, the number of points contained in this subtree, and a pointer to the child.

The insertion algorithm of SR-trees is based on that of SS-trees. The difference is the way SR-trees adjust bounding regions for the inserted point. SR-trees need to adjust both bounding spheres and bounding rectangles while SS-trees need to adjust only bounding spheres. When computing the bounding sphere of a parent node, SR-trees utilize both the bounding spheres and the bounding rectangles of their child nodes. This permits SR-trees

to produce smaller spheres than SS-trees do, and thus reduces the overlap of bounding spheres. The bounding rectangles of SR-trees are adjusted in the same way as R^*-trees.

In common with the R^*-tree and the SS-tree, the deletion algorithm of the SR-tree is the same with that of the R-trees.

2.1.4 Grid Files

The grid file [30] is a typical representative for the non-hierarchical access methods. It imposes intervals on each axis of the universal space and divides the space into grid cells. The central part of Figure 2.9 illustrates the grid partition. The partition is dynamic and can be modified in response to insertions and deletions of data points by means of splitting an interval or merging two adjacent intervals. Each grid points to a disk block which contains the data points situated in this grid cell. The disk blocks are referred to as "buckets" in grid files. All the points in one grid cell must be stored in the same bucket. Several grid cells may point to the same bucket as long as the resulting bucket region, which is the space spanned by the cells which share the bucket, is a n-dimensional rectangle. The regions of buckets are pairwise disjoint; together they span the space. The crucial data structure of the grid file is the grid directory, which consists of two parts. The first is a dynamic n-dimensional array, containing one entry for each grid cell. The values of the elements of this array are pointers to the relevant data buckets. The second part is a set of k one-dimensional arrays called linear scales; each scale defines a partition of a space axis. Figure 2.9 (from [18]) illustrates a grid file. The center of the figure is the grid directory. The data points are displayed in the directory for demonstration purposes only; they are actually stored in the buckets. Notice that in the lower left part there are four grid cells pointing to the same bucket.

To answer a query, one first uses the linear scales to locate the grid cells, and then searches the relevant buckets according to the pointers stored in the grid cells.

As data points keep coming, the buckets may overflow. The overflowing bucket is split and the data points are distributed between the resulting two buckets. When there is more than one grid cell pointing to the overflowing bucket, and the existing grid partition can be used for splitting the bucket successfully, then we need merely to allocate a new bucket and adjust the mapping between grid cells and buckets. Otherwise, we must refine the grid partition by splitting an interval on an axis. In this case, we have a choice with respect to the split dimension and the location of the splitting point. Without any external knowledge or motivation, a reasonable splitting policy is one that cycles through the various dimensions and uses interval midpoints. However, the splitting policy may favor one dimension over the others according to particular applications, and buckets may be split at points other than interval midpoints.

A well-known drawback of grid files is that the size of the grid directory grows too fast. When an interval is split, the number of grid cells is doubled. This results in a super-

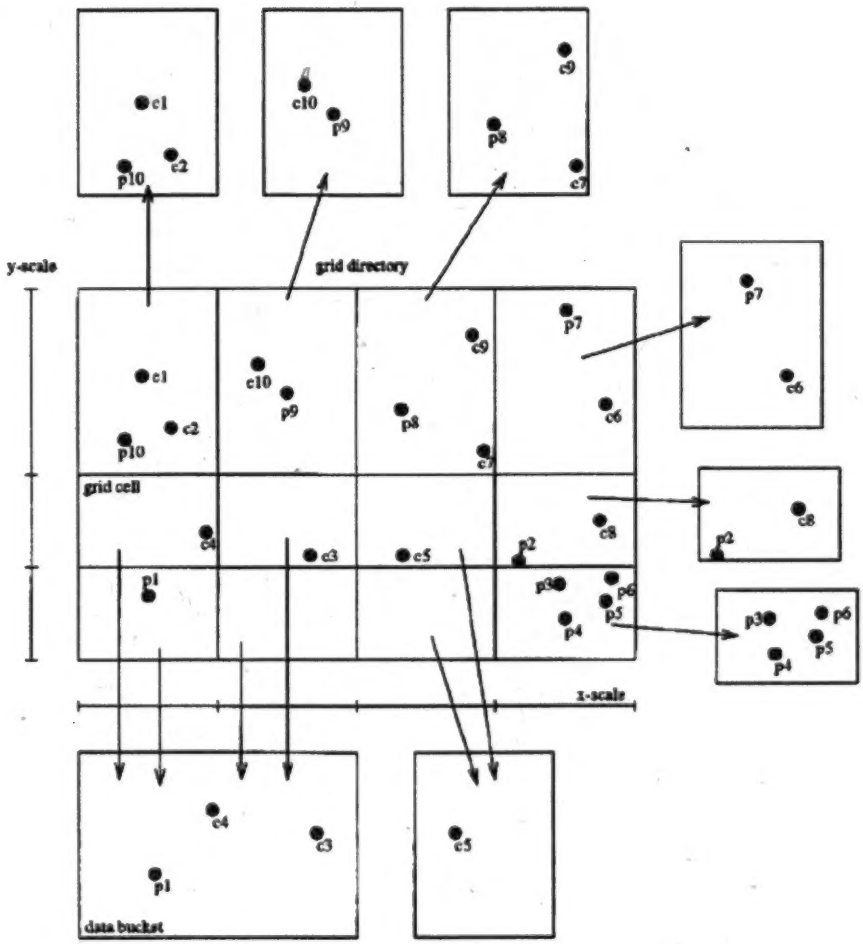


Figure 2.9: Grid file (from [18])

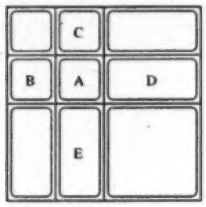


Figure 2.10: Buckets in grid file

linear directory growth even for uniformly distributed data. When the data distribution becomes less uniform, the directory expansion approaches an exponential rate. The problem is magnified by the number of dimensions of the space [15].

The counterpart of splitting is merging, which occurs when the bucket occupancy is too low. The decision as to which bucket pairs should be merged can be made based on a buddy system or a neighbour system. In a buddy system, each bucket—say X—can be merged with exactly one bucket in each of the n dimensions, so that the resulting bucket region can be obtained by a regular binary subdivision of the space. Ideally the chosen bucket—say Y—should have the property that at some earlier point some bucket was split to yield buckets X and Y. We call this buddy system "true buddy". In a neighbour system, each bucket can be merged with either of its two adjacent neighbours in each of the n dimensions, as long as the resulting bucket region is a n-dimensional rectangle. For example, the bucket A shown in Figure 2.10 can be merged with buckets B and C in the buddy system; in the neighbour system, bucket A can be merged with buckets B, C, D, and E.

2.2 Dimensionality Reduction and Filtering

The image feature vectors usually have a high number of dimensions. For example, colour histograms typically have at least 64 dimensions. However, it is well known that current multi-dimensional indexing structures suffer from "dimensionality curse", which refers to the phenomenon that the query performance of the indexing structures degrades as the data dimensionality increases. Moreover, Beyer et al. reported [4, 5] a "clustering" phenomenon: as dimensionality increases, the distance to the nearest data point approaches the distance to the farthest data point. The "clustering" phenomenon can occur for as few as 10-15 dimensions. Under this circumstance, high-dimensional indexing is not meaningful: linear scan can outperform the R*-tree, SS-tree and SR-tree [5]. Hence, developing more sophisticated multi-dimensional indexing structures is not a complete answer to the question of how to provide effective support for querying high-dimensional data. Reducing the dimensionality of the data is employed to complement the use of multi-dimensional index.

The dimensionality reduction problem is defined as: given a set of vectors in n-dimensional space, find the corresponding vectors in k-dimensional space ($k < n$) such that the distances between the points in the original space are maintained as well as possible. The following

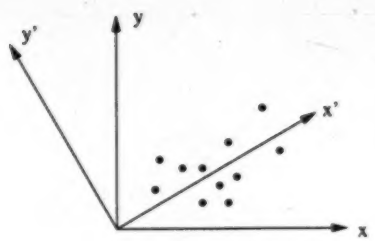


Figure 2.11: Illustration of the Karhunen-Loève transformation

stress function gives the average relative error that a distance in k-dimensional space suffers from:

$$stress = \sqrt{\frac{\sum_{i,j}(d'_{ij} - d_{ij})^2}{\sum_{i,j} d_{ij}^2}}$$

where d_{ij} is the distance between objects i and j in their original n-dimensional space and d'_{ij} is their distance in the resulting k-dimensional space. Preserving distances means minimizing the stress. There have been several techniques developed for dimensionality reduction, such as multi-dimensional scaling (MDS) [12], Karhunen-Loève (K-L) transform [12], and fast map [12]. The basic idea of multi-dimensional scaling is to first assign each object to a k-dimensional point arbitrarily; and then try to move it in order to minimize the discrepancy between the distances in the original space and those in the resulting space. The Karhunen-Loève transform chooses the most important k elements (actually linear combinations of feature elements) from the n-dimensional vectors by computing the eigenvectors of the covariance matrix[1] and projecting vectors on the k largest eigenvalues. This operation is closely related to the singular value decomposition[2] (SVD). Figure 2.11 illustrates the K-L transform. If the dimensionality is reduced to 1, K-L transform projects the 2-dimensional points shown in Figure 2.11 on the direction of x'. The fast map method has the advantage that, as the name suggests, it can be computed quickly. The heart of the fast map method is to project the objects onto a "line" defined by two carefully selected objects from the dataset. The heuristic of selecting objects is that the two objects should

[1]The covariance of two dimensions is a measure of their tendency to vary together, i.e., to co-vary. More precisely, consider a set of m examples of n-dimensional vectors $\{X_1, X_2, \ldots, X_m\}$. Let $X_l(i)$ denote the i-th element of vector X_l. Let $M(i)$ be the mean of the i-th dimension, and $M(j)$ be the mean of the j-th dimension. The covariance of the i-th dimension and the j-th dimension $c(i,j)$ is defined by

$$c(i,j) = \{[X_1(i) - M(i)][X_1(j) - M(j)] + \cdots + [X_m(i) - M(i)][X_m(j) - M(j)]\} \div (m-1).$$

The covariance has several important properties: if the i-th and the j-th dimensions tend to increase together, then $c(i,j) > 0$; if the i-th dimension tends to decrease when the j-th dimension increases, then $c(i,j) < 0$; if the i-th and the j-th dimensions are independent, then $c(i,j) = 0$. All of the covariances c(i,j) can be collected together to form a covariance matrix.

[2]Let A be an m-by-n matrix composed of m d-dimensional vectors. The singular value decomposition of A is the factorization $A = U\Sigma V^T$, where U is an m-by-n orthogonal matrix, V is an n-by-n orthogonal matrix, and $\Sigma = diag(\sigma_1, \ldots, \sigma_n)$ is an n-by-n diagonal matrix with $\sigma_1 \geq \cdots \geq \sigma_n \geq 0$. The transformed vector is computed by multiplying the original vector with the matrix V.

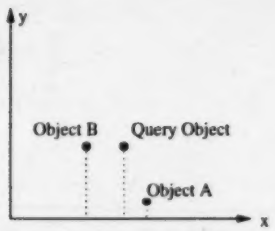


Figure 2.12: Distance ordering not preserved when projecting

be as far away from each other as possible.

The above techniques are only applicable to static databases where the set of data objects is known *a priori*. Kanth *et al.* propose techniques for performing SVD-based dimensionality reduction in dynamic databases [22]. When the data distribution changes considerably, due to inserts and deletes, the SVD transform is recomputed using an aggregate data set whose size is much smaller than the size of the database, in order to save computation overhead. The aggregate data are centroids of the clusters of database data. Centroids reflect the data distribution well, which ensures the quality of the recomputed SVD.

Dimensionality reduction techniques may result in the loss of information and the subsequent loss in query accuracy. To achieve 100% query accuracy, a filtering approach is adopted. The idea is to make use of the lower-dimensional vectors (obtained by dimensionality reduction techniques) to perform a course-grained, but efficient filtering. The fine-grained n-dimensional refinement only applies to those data that pass through the earlier filtering. The filter should have the property that it allows some false hits, but not false dismissals.

To ensure 100% accuracy, for range queries, the bounding relation $d_n \geq \lambda d_l$ between the distance in high-dimensional space d_n and the corresponding distance in low-dimensional space d_l must hold, where λ is a positive real number. Thus, the range query $d_n \leq \epsilon$ can be translated into low-dimensional space as $d_l \leq \frac{\epsilon}{\lambda}$, and the data objects retrieved using $d_l \leq \frac{\epsilon}{\lambda}$ are guaranteed to be a super set of the set retrieved using $d_n \leq \epsilon$. For example, Faloutsos *et al.* proved in [11] that quadratic colour histogram (high-dimensional vector) distance is bounded by average colour (three-dimensional vector) distance. Thus colour histogram filtering is achieved using average colours (see Section 3.3). In [21], the colour histogram filter that uses three-dimensional vectors is generalized through SVD techniques to use k-dimensional vectors.

Nearest neighbour queries, however, cannot be processed directly using filtering approaches because filters usually do not preserve ordering. It is possible that object A is closer to the query object than object B at the first level, but it is actually farther away from the query object than object B. For example, Figure 2.12 shows that this situation happens when two-dimensional points are projected onto the x axis. So, nearest neighbour queries have to be translated to range queries. If k nearest neighbours are wanted, the

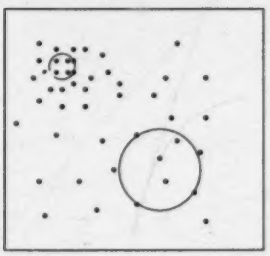


Figure 2.13: Small range in dense portion; large range in sparse portion

range threshold should be chosen so that at least k data objects are retrieved finally. The choice of the range threshold is critical. Ideally, the range query should retrieve exactly k data objects. If the chosen range threshold, say t, is too high and too many objects are retrieved, work is wasted in calculating the distances to the objects that we do not need; if, on the other hand, t is too low and the number of retrieved objects is less than k, the search has to restart with a larger threshold, say T, though we can issue a range query in the form of $t < d_l \leq T$ to avoid processing the same data object more than once. A good choice of range threshold should be based on the data distribution. If the query object is in the dense portion of the database, the range threshold should be relatively small; if the query object is in the sparse portion, the range threshold should be relatively large, as illustrated by Figure 2.13.

Multiple levels of filtering can be applied. In three-level filtering, for example, n-dimensional vectors are reduced to k_1-dimensional vectors and k_2-dimensional vectors, respectively, where $k_1 < k_2 < n$. Similarity searches are first conducted in k_1-dimensional space; the retrieved object is then carried over to the k_2-dimensional space; finally the n-dimensional distance metric is applied. Ng *et al.* discuss the issue of applying multi-level filtering [28]. By analyzing the CPU and I/O costs, they conclude that three-level structures are typically the best, and structures that have at least four levels are not promising.

Zhang *et al.* propose [45] a very different approach to filtering. They do not utilize low-dimensional spaces; instead, they use an artificial neural network—the Self-Organization feature Map (SOM)—to automatically build indexing trees by abstraction. In the retrieval process, unlikely candidates are eliminated as one goes down the indexing tree.

2.3 Indexing in Metric Spaces

Besides indexing data objects in vector spaces, the indexing problem can be approached from a rather different perspective, that is, indexing in metric spaces. In metric spaces, how data objects are defined is not important (data objects may or may not be defined as vectors.); what is important is the definition of the distance between data objects. The distance function $d(x, y)$ must have the following properties:

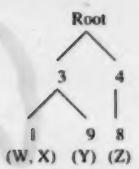


Figure 2.14: An example trie with four objects and two keys (from [3])

1. $d(x, y) = d(y, x)$ (symmetry),
2. $0 \leq d(x, y) < \infty$, and $d(x, y) = 0$ iff $x = y$ (non negativity),
3. $d(x, y) \leq d(x, z) + d(z, y)$ (triangle inequality).

Berman proposes using triangulation tries [3] to index in metric spaces. The idea is to choose a set of *key objects* (key objects may or may not be in the datasets to be indexed), and for each object in the dataset, create a vector consisting of the ordered set of distances to the key objects. These vectors are then combined into a trie. The trie [14] is a tree-like structure. The number of levels in the trie is one more than the number of key objects; each level corresponds to a key object, except the leaf level. The number of branches at each level is equal to the number of different values of distance from the objects to be indexed to the corresponding key object. An example from [3] clearly illustrates this idea. Suppose there are four objects W, X, Y, and Z in the dataset and there are two key objects, k_1 and k_2. The distances from W to k_1 and k_2 are 3 and 1, respectively, which form a vector $v_W = (3, 1)$. Similarly, we have $v_X = (3, 1)$, $v_Y = (3, 9)$, and $v_Z = (4, 8)$. These vectors are combined in a trie shown in Figure 2.14.

Space decomposition is another approach to indexing in metric spaces. J. K. Uhlmann outlines in [39] two different methods of space decompositions. One is named *ball decompositions*, which breaks the metric space up by using sphere cuts. Ball decomposition builds a binary tree. It picks an arbitrary object as a root of the tree and calculates the radius of a median sphere centered at the object, so that half the remaining objects fall inside the sphere—i.e., the distances from these objects to the centering object are smaller than the sphere radius. The objects which are inside the sphere are put in one branch of the tree and those which are outside the sphere are put in another branch. The lower level branches of the tree are constructed recursively in the same manner.

The other decomposition method proposed in [39] is named *generalized hyper-plane decompositions*. A generalized hyper-plane is defined by two objects o_1 and o_2 and consists of the set of objects p satisfying $d(p, o_1) = d(p, o_2)$. An object x is said to lie on the o_1-side of the plane if $d(p, o_1) < d(p, o_2)$. The generalized hyper-plane decompositions builds a binary tree. At the root node, two arbitrary objects are picked to form a hyper-plane. The objects which are at the one side of the hyper-plane are placed in one branch of the tree, and those at the other side of the hyper-plane are placed in the other branch. The lower

level branches of the tree are constructed recursively in the same manner.

Brin [7] proposes GNAT, standing for Geometric Nearneighbour Access Tree. The decomposition method used in GNAT is a generalization of the "generalized hyper-plane" method. It first randomly chooses k data objects from the dataset to form a root node of the tree. These objects are referred to as *split objects*. The remaining objects in the dataset are associated with the closest split objects, and thus form the k branches of the tree. The lower level branches are then constructed recursively in the same manner.

The above decomposition methods build trees by a top-down recursive process, so the trees are not guaranteed to remain balanced in case of insertions and deletions. Furthermore, these methods do not consider secondary memory management, so they are not suitable for large databases which have to be stored on disks. To address these problems, M-trees [44, 8] are proposed. M-trees are dynamic balanced trees. Each node of M-trees corresponds to a disk block. Like the ball decompositions proposed in [39], M-trees use sphere cuts to break up the metric space. The difference is that M-trees are multi-branch trees and are constructed in a bottom-up fashion. All data objects are stored in leaf nodes. When a leaf node overflows, it is split. Two objects, referred to as *routing objects*, are promoted to the parent node. Each routing object is associated with a pointer pointing to one of the nodes resulting from the split, which is propagated up the tree. The node split propagation may cause the root to split. In this case, a new root node is created. A routing object serves as the center of a *covering sphere*, so that the objects in the subtree under the routing object all fall into the sphere; that is, the distance from the routing object to any object in the subtree is smaller than the radius of the covering sphere. When inserting a data object, the insertion algorithm chooses the sub-tree whose *covering radius* (the radius of the covering sphere) needs the least increase to accommodate the new object. If multiple subtrees with this property exist, the one whose routing object is closest to the new object is chosen. When a node needs to be split, several split policies are possible [8]; the basic rules are to minimize the covering radius, minimize the "overlapping region", and distribute objects evenly between the two resulting nodes.

Metric space decompositions are made based on distance measures from some reference objects in datasets. The use of dataset elements in defining partitions tends to permit exploitation of the distribution features of the dataset itself, and thus may provide good query performance.

Indexing in metric spaces requires nothing to be known about the objects other than their pairwise distances. It only makes use of the properties of distance measures (symmetry, non-negativity, triangle inequality) to organize the objects and prune the search space. Thus it can deal with objects whose topological relationships are unknown.

Chapter 3

Colour Histogram Comparisons

In this chapter, the use of colour histograms in judging image colour similarity is discussed. We first briefly introduce colour spaces and colour histograms. Then we examine colour histogram distance metrics. We will also describe multi-scale representation of images with colour histograms, and discuss the application of multi-scale representation to multi-precision similarity queries and sub-image similarity queries.

3.1 Colour Spaces

A colour space is a three-dimensional definition of a colour system. The identifying attributes of the colour system are mapped onto the coordinate axes. Colours are represented as points in the three-dimensional colour spaces. Many different colour spaces exist, such as RGB, $L^*u^*v^*$, and HSV [13, 6]; they each have advantages and disadvantages for colour selection and specification.

RGB is the most commonly used colour space. It is an additive colour space; that is, colours in RGB space are obtained from the addition of the three primaries—Red, Green, and Blue. The greater the red, green, or blue values, the brighter the colour. The RGB colour space employs a Cartesian coordinate system and is represented as a unit cube defined on R, G, and B axes, as shown in Figure 3.1. The line that runs from Black (0,0,0) to White (1,1,1) is the greyscale line. In order to have an achromatic (colourless) pixel, the values of R, G, and B must be the same.

The perceptive measures of coloured light are hue, saturation, and lightness. Hue is a colour attribute that describes a pure colour, such as red, blue, or yellow. Saturation gives a measure of the degree to which a pure colour is diluted by white light. Colours become grey if completely unsaturated. Lightness is the perceived intensity of reflected light; it is based on perceptual measures of brightness, instead of physical measures. The HSV space is defined to meet humans' intuition of perception of colours. In HSV colour space, a colour is specified by three parameters: hue, saturation, and value (analogous to lightness). The three-dimensional representation of the HSV space is derived from the RGB space cube. If we look at the RGB cube along the grey diagonal we can see a hexagon. The HSV space

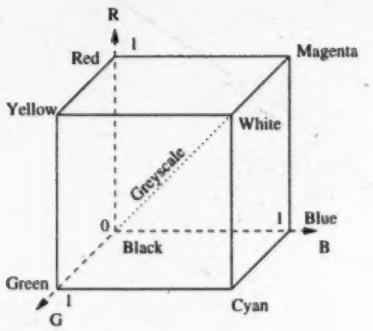


Figure 3.1: The RGB colour space cube

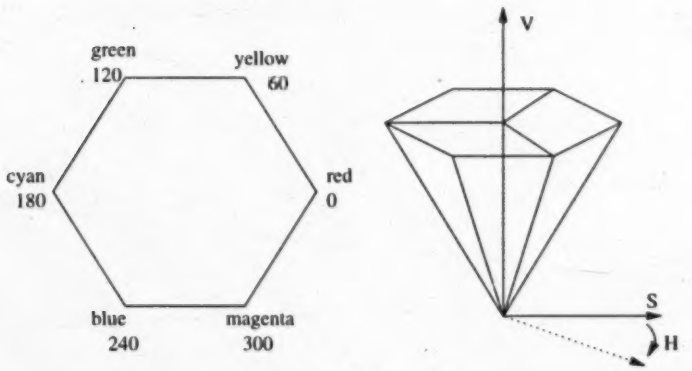


Figure 3.2: The HSV colour space: hex-cone

is a hex-cone (six-sided pyramid turned upside down) and employs a cylindrical coordinate system, as shown by Figure 3.2. The hue is given by the angle around the vertical axis with red at 0, yellow at 60, green at 120, cyan at 180, blue at 240, and magenta at 300. Saturation ranges from 0 along the centerline of the hex-cone, to 1 at the side of the hex-cone. The saturation measures the ratio of purity; saturation equal to 1 indicates the maximum purity. Value ranges from 0 at the bottom point of the hex-cone, to 1 at the top.

A desirable property of colour space is that the geometric distance between two colours is proportional to the difference perceived by a human observer (perceptual difference). This property is termed as *perceptual uniformity*. A *uniform* colour space is a space such that a colour difference perceived by a human observer is approximated as the distance between two points in the colour space.

The $L^*u^*v^*$ colour space [13, 6] appears as an attempt to create a standard uniform colour space; it makes easier the evaluation of perceptual distances between colours. L axis represents the lightness; u and v are chromatic coordinates. Figure 3.3 shows the visible colour gamut (curved shape) in (u^*,v^*) perceptually uniform coordinates. The visible spectrum starts with blue at the bottom of the graph, moving through green in the upper

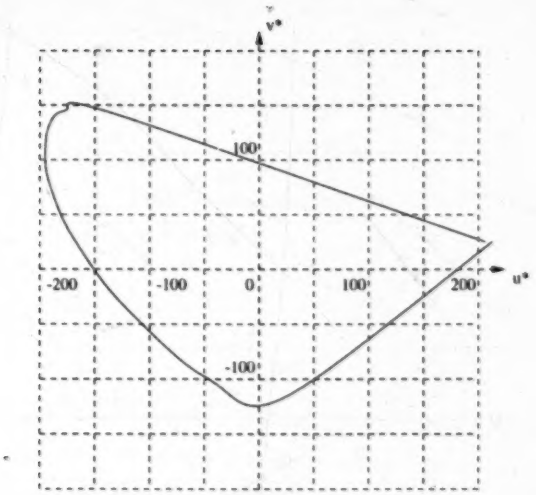


Figure 3.3: Colour gamut in u^*v^* coordinate system

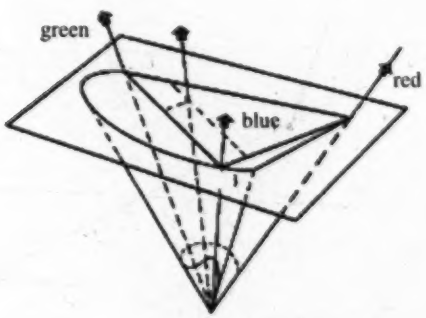


Figure 3.4: A three-dimensional representation of $L^*u^*v^*$ colour

left, and out to red in the upper right. Figure 3.4 is a three-dimensional representation of $L^*u^*v^*$ colour space: it is roughly an inverted cone. In $L^*u^*v^*$ space, the perceptual distance between two colours is approximated by the Euclidean distance as follows

$$d^2 = (\delta L^*)^2 + (\delta u^*)^2 + (\delta v^*)^2.$$

3.2 Colour Histograms

A colour histogram is the most traditional way of describing low-level colour properties of images. An image colour histogram is a discrete function $h(c_k) = n_k$, where c_k is the k-th colour value and n_k is the number of pixels in the image with that colour. In order to compare colour histograms of images with different sizes, colour histograms are often normalized as $H(c_k) = \frac{n_k}{n}$ where n is the total number of pixels in the image.

Obviously, computing colour histograms requires selection and quantization of the colour

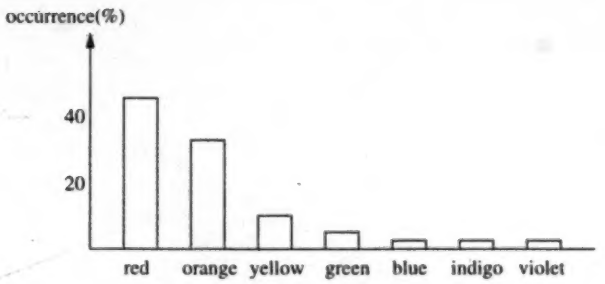


Figure 3.5: Histogram of an image in which warm colours are dominant

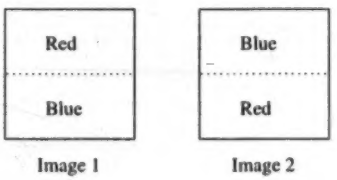


Figure 3.6: Two images with different colour layouts have the same colour histogram

space, and there has been a large variety of methods proposed. For example, Pass *et al.* uniformly quantize RGB colour space to 64 colours in their image retrieval system [17]. Gray, in order to retrieve images, chooses $L^*u^*v^*$ space and partitions it to 512 colours [19]. Smith *et al.* chooses HSV colour space and quantizes it to 166 colours [36]. However, there has been no consensus about which colour space is most suitable for colour histogram-based image retrieval. The problem is a result of the fact that there does not exist a universally accepted colour space definition, and colour perception is extremely subjective [41, 16]. Nonetheless, after selecting and quantizing the colour space, colour histograms can be computed simply by counting how many pixels belong to each colour.

The colour histogram indicates the distribution of colours in an image. Roughly speaking, colour histograms give an estimate of the probability of occurrence of discrete colour values. A plot of colour histogram function for all these colour values provides a global description of the appearance of an image. For example, a histogram plotted in Figure 3.5 indicates that the warm colours are dominant in the image.

Histograms, by themselves, do not include spatial information so that images with very different colour layouts can have similar colour histograms. For example, the two different pseudo-images shown in Figure 3.6 have exactly the same colour histogram. The multi-scale colour histograms, which will be discussed in Section 3.4, involve spatial information to some degree, and, therefore, provide better discrimination potential.

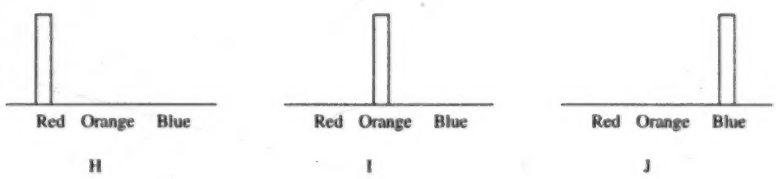


Figure 3.7: Colour Histograms whose perceptual similarities are not consistent with their L_1 or L_2 distance

3.3 Colour Histogram Distance Metrics

Colour histogram distances are used to measure colour dissimilarity between images. Colour histogram distance metrics are used to answer the question: "given two n-dimensional colour histograms $X(x_1, x_2, \ldots, x_n)$ and $Y(y_1, y_2, \ldots, y_n)$, how is the distance between them computed?"

One of the commonly used distance metrics is the L_1 norm, which computes the distance d between two n-dimensional colour histograms $X(x_1, x_2, \ldots, x_n)$ and $Y(y_1, y_2, \ldots, y_n)$ as:

$$d_{L_1}(X,Y) = \sum_{i=1}^{n} |x_i - y_i|.$$

A method called histogram intersection [38] is proposed to measure the similarity s between colour histograms as

$$s(X,Y) = \frac{\sum_{i=1}^{n} \min(x_i, y_i)}{\sum_{i=1}^{n} y_i}.$$

If the two colour histograms have the same size, i.e., $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$, then histogram intersection is equivalent to L_1 distance, shown as follows:

$$1 - s(X,Y) = \frac{1}{2T} \sum_{i=1}^{n} |x_i - y_i|$$

where $T = \sum_{i=1}^{n} x_i$. The other commonly used distance metric is the L_2 norm, which is computed as:

$$d_{L_2}(X,Y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}.$$

L^1 and L^2 distance metrics sometimes demonstrate poor performance because they do not take the colour similarity between colour histogram bins into account. For example, consider a histogram distribution of three colours, say red, orange, and blue, where red is considered to be more similar to orange than blue. Suppose there are three colour histograms shown in Figure 3.7. Apparently colour histogram H is perceptually closer to I than J. However, by L_1 or L_2 metric, the distances between these three colour histograms are equal.

A quadratic form distance function, referred to also as weighted Euclidean distance, which considers the histogram bin colour similarity, is defined in [21] as

$$d^2_{hist}(X,Y) = Z^T A Z$$

where $Z = (X - Y)$, Z^T denotes the transpose of Z, and $A = [a_{ij}]$ is a *similarity matrix* whose elements a_{ij} denote similarity between colours i and j. The larger a_{ij}, the more similar are colours i and j. These weights can be normalized so that $0 \leq a_{ij} \leq 1$ and $a_{ii} = 1$. One way of defining a_{ij} can be:

$$a_{ij} = (1 - d_{ij}/d_{max})$$

where d_{ij} is the Euclidean distance between colours i and j in some colour space; $d_{max} = max_{i,j}(d_{ij})$. If the RGB colour model in which R, G, and B values range from 0 to 255 is used, $d_{ij} = \sqrt{(R_i - R_j)^2 + (G_i - G_j)^2 + (B_i - B_j)^2}$ and $d_{max} = \sqrt{3 \times 255^2}$.

When colour histograms are normalized so that $0 \leq x_i, y_i \leq 1$ and $\sum_i x_i = \sum_i y_i = 1$, the colour histogram distance can also be defined in a $(n-1)$-dimensional space. Decompose the matrix A as follows:

$$A = \begin{bmatrix} A_{n-1} & a_{*n} \\ a_{*n}^T & a_{nn} \end{bmatrix}$$

where A_{n-1} is the top left $(n-1) \times (n-1)$ elements and a_{*n} is the nth column of A less the last entry (a_{nn}). Define $\tilde{A}$ as

$$\tilde{A} = [A_{n-1} - a_{*n} \cdot 1^T - 1 \cdot a_{*n}^T + a_{nn} 1 \cdot 1^T]$$

where 1 is a vector of $(n-1)$ ones. Let $\tilde{Z}$ be a vector formed by the first $n-1$ elements of Z. The colour histogram distance can be defined in $n-1$ dimensional space as

$$d_{hist}^2(X,Y) = \tilde{Z}^T \tilde{A} \tilde{Z}$$

Although the full comparison between n-dimensional colour histograms can be reduced into a $(n-1)$-dimension without loss of accuracy, it is still computationally expensive. A 3-dimensional compact representation of colour histograms—average colours— is proposed as a cheaper way to simulate full n-dimensional histogram comparisons [21]. The idea is to filter the database using average colour comparisons first, and then apply full colour histogram comparisons to the set of images which are retrieved at the filtering step.

Given that each bin of a colour histogram represents a three-dimensional colour value, the average colour of a colour histogram is defined to be the weighted average colour corresponding to the normalized colour histogram. Specifically, let $C = [c_1 c_2 \ldots c_n]$ be a $3 \times n$ matrix whose i-th column is the colour $c_i = [\alpha_i \beta_i \gamma_i]^T$, where α, β, and γ represent the magnitudes along the three colour dimensions. Given two normalized n-dimensional colour histograms, X and Y, the 3×1 average colour vector for each is

$$X_{ave} = CX, \; Y_{ave} = CY.$$

The squared average colour distance is defined by

$$d_{ave}^2 = (X_{ave} - Y_{ave})^T (X_{ave} - Y_{ave}) = (X - Y)^T C^T C (X - Y).$$

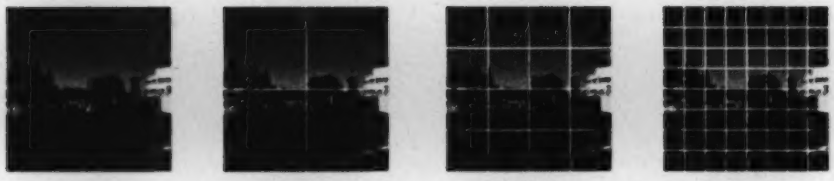


Figure 3.8: The 4-level multi-scale representation

If we let $W = C^T C$, the above expression can be rewritten as

$$d_{ave}^2 = Z^T W Z = \tilde{Z}^T \tilde{W} \tilde{Z}$$

where $\tilde{W}$ is defined in terms of W in the same way that $\tilde{A}$ is defined in terms of A.

While the average colour comparisons are not as accurate as one between full n-dimensional histograms, they are much faster. Moreover, the images retrieved by average colour comparisons are guaranteed to include all images that should be retrieved by colour histogram comparisons. This is stated in the following theorem [21]:

> "With d_{hist} and d_{ave} defined as above, if $\tilde{A}$ is positive semidefinite, then for all vectors X and Y, $d_{hist}^2 \geq \lambda_1 d_{ave}^2$, where λ_1 is the minimum eigenvalue of the generalized eigenvalue problem $\tilde{A}\tilde{z} = \lambda \tilde{W} \tilde{z}$".

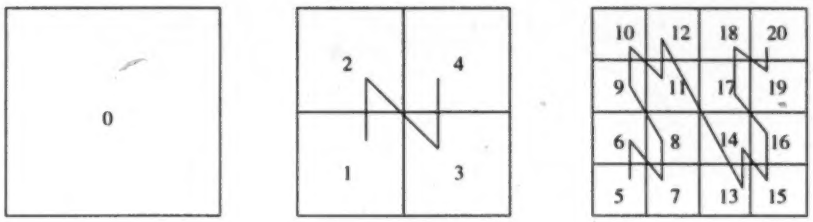
According to the above theorem, for any range query of the form $d_{hist} \leq \epsilon$, $d_{ave} \leq \epsilon / \sqrt{\lambda_1}$ can be used to retrieve images quickly and without misses. The expensive measure d_{hist} will then have to be applied only to the filtered set of images.

The average colour distance also provides a method of indexing images. Instead of indexing colour histograms, which are high-dimensional data, an index structure can be built on 3-dimensional average colour. In this way, we avoid the "dimensionality curse". After the filtering step, linear scan can be performed over the retrieved images to apply colour histogram comparisons.

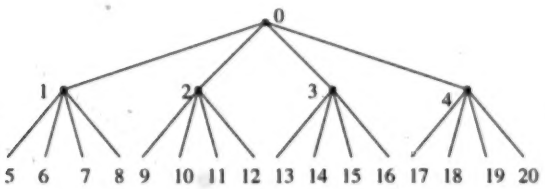
3.4 Multi-scale Colour Histograms

Besides extracting colour histograms from entire images, an image can also be segmented into several blocks, each of which has an associated colour histogram. Figure 3.8 shows a 4-level multi-scale representation in which the entire image is divided into four blocks, and each block is recursively divided into four, and so on. A colour histogram is computed for each of the blocks at each level. These colour histograms together form *multi-scale colour histograms* of an image.

A suitable structure for storing these colour histograms is a quadtree. A quadtree is a tree whose nodes are either leaves or have 4 children. The root of the tree corresponds to the entire image; the children of a node represent the four quadrants. Each of the tree



(a) N order of image blocks



(b) quadtree with predetermined children node order

Figure 3.9: A quadtree stores colour histograms of image blocks

node stores its corresponding block's colour histogram. The order of the children nodes are predetermined. For example, when an N order shown in Figure 3.9(a) is used, the corresponding quadtree is shown in Figure 3.9(b).

If the quadtree is implemented as a hierarchical data structure, then each of the tree nodes should have at least four pointers which point to its four child nodes. To eliminate the need for pointers, the quadtree can be implemented as a linear array. A quadtree implemented in this way is referred to as a linear quadtree. The tree nodes line up in an array in the order specified in Figure 3.9. The first node (with index 0) in the array is the root of the tree. Given a node with index i, its four child nodes' indices are $(i \times 4) + 1$, $(i \times 4) + 2$, $(i \times 4) + 3$, and $(i \times 4) + 4$, and its parent node's index is $\lfloor (i - 1) \div 4 \rfloor$. Thus the linear quadtree not only represents the quadtree in a more compact way, but also provides a direct access to every node in the quadtree.

3.5 Multi-precision Similarity Queries

With the multi-scale colour histograms, image similarity queries based on colour histograms can be done at several levels of precision. In a four-level decomposition, two images can be similar at four precision levels:

1. at the first level, the colour histogram of the entire image is compared: 1 colour histogram comparison;

2. at the second level, the colour histograms corresponding to the 4 blocks ($\frac{1}{4}$ of the entire image) are compared: 4 colour histogram comparisons;

3. at the third level, the colour histograms corresponding to the 16 blocks ($\frac{1}{16}$ of the entire images) are compared: 16 colour histogram comparisons;

4. at the fourth level, the colour histograms corresponding to the 64 blocks ($\frac{1}{64}$ of the entire image) are compared: 64 colour histogram comparisons.

At the first level, the image distance d is defined as the distance between colour histograms of entire images. At the higher levels, d is defined as the average of the colour histogram distances computed for all the blocks. At the fourth level, for instance, $d = \frac{1}{64}\sum_{i=1}^{64} d_i$, where d_i is the colour histogram distance computed for the i-th image block. This image distance metric not only takes the image colour composition into account, but also captures the spatial distribution of colour in an image. Thus, the distance metrics at the higher precision level provide better discrimination power.

If images are similar at a higher level, they must be similar at the lower levels too, in order to make sense. For this reason, the colour histogram distance must be formulated in such a way that the image distance at the lower level is smaller than the distance at the higher level, i.e., $d_I \leq d_{II} \leq d_{III} \leq d_{IV}$ where d_I is the distance between two images at the first level, d_{II} is the distance between the same two images at the second level, and so on. In Appendix A, we prove that $d_I \leq d_{II} \leq d_{III} \leq d_{IV}$ holds for the weighted Euclidean distance metric.

Obviously, the distance metrics defined at higher precision levels are much more computationally expensive than those defined at lower levels. Fortunately, with the property of $d_I \leq d_{II} \leq d_{III} \leq d_{IV}$, efficient multi-scale search strategies with the use of lower level distances as filters can be explored. As suggested in [24], filtering schemes should start with the first level, and not skip any intermediate levels. So combined with the average colour filtering, when the fourth precision level query of the form $d_{IV} \leq \epsilon$ is processed, the following filtering is applied in turn: $d_{ave} \leq \epsilon/\sqrt{\lambda_1}$ (d_{ave} and λ_1 are described in Section 3.3), $d_I \leq \epsilon$, $d_{II} \leq \epsilon$, $d_{III} \leq \epsilon$, and $d_{IV} \leq \epsilon$. When the third precision level query is processed, the filtering stops at $d_{III} \leq \epsilon$; when the second precision level query, is processed, the filtering stops at $d_{II} \leq \epsilon$, and so on.

There are basically three strategies for using the filtering scheme: PV (Pure Vertical), PH (Pure Horizontal), and HV (Horizontal-and-Vertical) [24]. PV strategy takes an image from the database and proceeds toward the higher precision level until either the image fails at a particular level (i.e., the distance is so large that the image cannot be qualified as a good match) or the highest level is reached. Then the next image is taken from the database and processed the same way. PV strategy checks the database "vertically": it finishes with an image before it starts to check another one. PH strategy, on the contrary, starts with checking all the database images at the lowest level. The qualified images are carried over to the next level until the highest level is reached. PH strategy checks the database "horizontally": it finishes checking all the images at a precision level before it starts to check the next level. HV is a hybrid of PV and PH. At the lowest level, the PH

strategy is applied to eliminate poor matches. The qualified images are then carried to the next stage, in which the PV strategy is used. Figure 3.10 (adapted from [24]) illustrates these three strategies.

Data file organization decides which search strategy should be used. The filtering strategy should check data in the same order as they are organized in files in order to avoid frequently jumping back and forth in files, which imposes a high I/O cost. If the colour histograms of quadrants of the same image are stored continuously in a linear quadtree, the PV strategy is favored because it has a lower I/O cost. Furthermore, if an index on the average colours is built to avoid inefficient sequential scanning, the average colours of all images must be checked first, which results in the HV strategy shown in Figure 3.10(c).

3.6 Sub-image Similarity Queries

Some users may not remember the contents of the whole image, or they do not care about certain parts of an image. In this case, users provide only the colour information of the part of the image they remember or care about, in order to issue a similarity query. We call these kind of queries sub-image similarity queries.

Multi-scale colour histograms can be used to support sub-image queries. The fourth level of decomposition imposes an 8-by-8 grid on an image (see Figure 3.8, the 4th level). Users can specify the part of image in which they are interested by choosing any of the grid cells. Several grid cells can be chosen at the same time, as long as the resulting shape is a rectangle. The selected rectangle is composed of predefined image quadrants, each of which is uniquely identified by a number (see Figure 3.9)—the selected rectangle can be identified by a set of numbers. For example, the region in Figure 3.11 is identified by { 11, 17, 34, 36, 58, 60 }.

The colour histogram of the selected portion can be computed based on the precomputed multi-scale colour histograms. For example, if the solid inner rectangle in Figure 3.11 is chosen as the portion of interest, its colour histogram can be computed from two third-level colour histograms (part 1 and part 2) and four fourth-level colour histograms (part 3 to part 6). Assuming that the colour histograms are normalized, the computation formula is

$$H = \frac{1}{3}H_{11} + \frac{1}{3}H_{17} + \frac{1}{12}(H_{34} + H_{36} + H_{58} + H_{60})$$

where H_{11}, H_{17}, ..., H_{60} are colour histograms of the corresponding numbered parts.

The user-provided example for sub-image query can be either a whole image or a block of a whole image. The block of a whole image can be any rectangle inside the image, not necessarily aligned to the 8-by-8 grid cells. The reason for this flexibility is that unlike the images stored in databases, there is no precomputed colour histograms for the example image. Regardless of how the portion is specified, its colour histogram has to be computed from scratch by counting pixels.

Level/Image No.	1	2	3	4	5	6	7	⋯
d_{ave}	•	•	•	•	•	•	•	⋯
	↓	↓	↓	↓	↓	↓	↓	↓
d_I	•	•	•	•	•	•	○	⋯
	↓	↓	↓	↓	↓	↓	↓	↓
d_{II}	•	•	•	○	•	○	○	⋯
	↓	↓	↓	↓	↓	↓	↓	↓
d_{III}	•	•	○	○	•	○	○	⋯
	↓	↓	↓	↓	↓	↓	↓	↓
d_{IV}	•	○	○	○	•	○	○	⋯

(a) PV Strategy

Level/Image No.	1	2	3	4	5	6	7	⋯
d_{ave}	• →	• →	• →	• →	• →	• →	• →	⋯
d_I	• →	• →	• →	• →	• →	• →	○ →	⋯
d_{II}	• →	• →	• →	○ →	• →	○ →	○ →	⋯
d_{III}	• →	• →	○ →	○ →	• →	○ →	○ →	⋯
d_{IV}	• →	○ →	○ →	○ →	• →	○ →	○ →	⋯

(b) PH Strategy

Level/Image No.	1	2	3	4	5	6	7	⋯
d_{ave}	• →	• →	• →	• →	• →	• →	• →	⋯
d_I	•	•	•	•	•	•	○	⋯
	↓	↓	↓	↓	↓	↓	↓	↓
d_{II}	•	•	•	○	•	○	○	⋯
	↓	↓	↓	↓	↓	↓	↓	↓
d_{III}	•	•	○	○	•	○	○	⋯
	↓	↓	↓	↓	↓	↓	↓	↓
d_{IV}	•	○	○	○	•	○	○	⋯

(c) HV Strategy

• Visited ○ Unvisited

Figure 3.10: Three search strategies (adapted from [24])

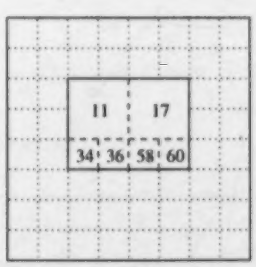


Figure 3.11: Specify portion of interest on the 8-by-8 grid

Even though the two image blocks to be compared are different in size, their normalized colour histograms can be compared directly. For the non-normalized colour histograms, the two image blocks have to be of the same size. Leung *et al.* [24] have developed Padding and Reduction algorithms to resolve the size differential between images blocks. They assume the example image is always smaller than the database image. They enlarge the example image by padding pixels, or reduce the database image by removing pixels in such a way that the estimated colour histogram distance is minimized. They suggest that the padding algorithm is better than the reduction algorithm when size differential is small, and vice versa when the differential is large. The reduction algorithm, however, requires colour histograms of database images to be "dynamic", i.e., colour histograms have to be changed according to the user-provided colour histogram before they are compared, which makes indexing database images extremely difficult, if not impossible.

Chapter 4

Multi-dimensional Extendible Hashing

Hashing is a technique that provides $O(1)$ access to data files. This is accomplished by computing a *hash function*. The hash function maps a *key space* to an *address space*, where the key space is the set of all possible key values, and the address space is the set of all storage units in a file. In other words, the hash function takes the key of a record and produces the address from which the record can be found. The computed address is called the *hash address*.

In this chapter, the hashing structure that was developed for indexing average colours of images or image quadrants is described. The range search algorithm that was designed based on the hashing structure is presented. The hashing structure can be extended to handle multi-dimensional data. The other extension is to apply the crucial component of the hashing structure to linear hashing, which results in another new index structure — multi-dimensional linear hashing.

4.1 Three-dimensional Extendible Hashing

The data record to be indexed by the extendible hashing is (average colour, image ID, image quadrant ID), where the average colour serves as the key. So the key space is the set of all possible colour values. The images or image quadrants that have the same average colour will share the same hash address. This requires that multiple data records be stored in the same storage unit. The storage unit, identified by a hash address, that can store multiple data records is referred to as *bucket*. Since the index is meant for similarity match, and not exact match, it is desirable that similar colours are hashed to the same bucket. Assume that the RGB colour model, in which each of the R, G, B values ranges from 0 to 255, is used. The following hash function divides the colour space into 64 cubes and the colours in the same cube is hashed to the same bucket. In other words, the hash function maps the 256^3 colour values into 64 hash addresses:

$$H(R,G,B) = \lfloor R \div 64 \rfloor \times 16 + \lfloor G \div 64 \rfloor \times 4 + \lfloor B \div 64 \rfloor$$

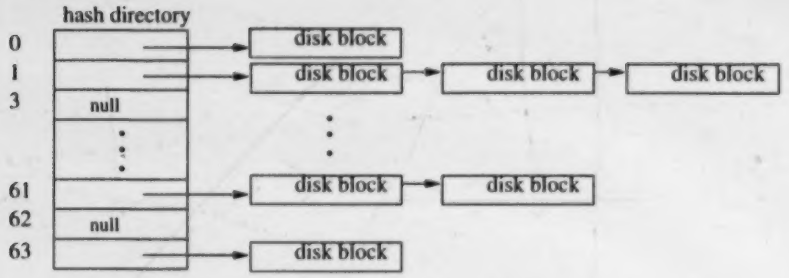


Figure 4.1: Hashing average colour into buckets

The R, G, B values are represented as 8-bit binary numbers. The above hash function can be computed quite fast by extracting the two most significant bits from each of the R, G, B values and then concatenating them. For example, given a colour value (00111100, 10101000, 01011001), the corresponding hash address is 001001 in binary number. Figure 4.1 illustrates the file structure. The number of directory entries is the size of the hash address space. Each entry in the hash directory corresponds to a bucket. If there are some data hashed to this bucket, the entry points to a disk block; otherwise the entry value is null. If one disk block is not enough to hold data records, several disk blocks are chained together.

Using the file structure shown in Figure 4.1 to index image average colours is not efficient, because the distribution of average colours is very skewed. As files increase in size, some of the hash directory entries may have hundreds of disk blocks chained together, while other entries may be empty. It is desirable that the size of address space, i.e., the size of hash directory can grow when new disk blocks are allocated, so that each directory entry contains only one disk block. The traditional extendible hashing technique [10] is capable of expanding and contracting the hash address space as needed, but it can only handle one-dimensional data. The new index structure, named three-dimensional extendible hashing, is an extension of the traditional extendible hashing to handle three-dimensional data. Three-dimensional extendible hashing is suitable for any colour model defined in a Euclidean coordinate system, e.g., the RGB and $L^*u^*v^*$ models (see Section 3.1). The RGB model, in which each of the R, G, B values ranges from 0 to 255, is used to describe the hash structure.

Traditional extendible hashing uses the leading d bits of the keys as hash addresses, and the size of address space is 2^d. If we consider the leading d bits of a key as a binary number b, then the bucket whose address is i ($0 \leq i \leq 2^d - 1$) contains all records for which b takes the value i. For example, if $d = 3$, the bucket with address 001 contains all records whose keys take the form 001 .... The value d is referred to as the *depth* of the hash directory. The value of d increases when the hash directory grows. Each disk block has a value p called the *local depth*. Local depth p means that all record keys in the disk block have common p

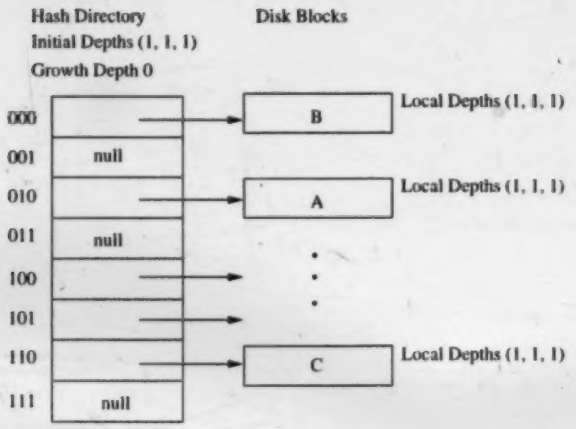


Figure 4.2: Initial hash directory of the three-dimensional extendible hashing

leading bits. When the disk block overflows and needs to be split, the $(p+1)$-th bit of the key is used to distribute the data records between the resulting two disk blocks, and the local depth of both the resulted disk blocks will be $p+1$. Initially, the local depth of disk block p equals the directory depth d. As address space increases in size (d increases), the local depth of some disk blocks will be less than d.

The concept of depth in traditional extendible hashing also applies to three-dimensional hashing. The hash directory of three-dimensional extendible hashing has three *initial depths* (d_1, d_2, d_3), one for each of the R, G, B colour components; it also has a *growth depth* d_g, which is 0 at the beginning and will increase as the address space increases. The number of bits of a hash address is $(d_1 + d_2 + d_3 + d_g)$, so the hash directory has $2^{(d_1+d_2+d_3+d_g)}$ entries. The disk block in three-dimensional hashing has three local depths (p_1, p_2, p_3), which means that all records in the disk block pointed to by the entry have common p_1, p_2, p_3 leading bits of the R, G, B values, respectively. At the beginning, the local depths of disk blocks are the same as the initial depths of the directory. The initial hash directory has $2^{(d_1+d_2+d_3)}$ entries since d_g is zero at the beginning. The $(d_1+d_2+d_3)$-bit hash address is computed by taking the leading d_1, d_2, d_3 bits from the R, G, B values, respectively, and then concatenating them. We call this $(d_1 + d_2 + d_3)$-bit address the *initial hash address*. For example, if the initial depths of a hash directory are (1, 1, 1), the initial hash address of colour value (R, G, B), represented as binary numbers (01111100, 10101000, 00011001), is a 3-bit binary number 010, which is computed by extracting the most significant bits from each of the colour components and then concatenating them. Figure 4.2 shows an initial hash directory whose initial depths are (1, 1, 1). The initial depths of the hash directory can be chosen according to the applications. If green colours are dominant and red colours barely appear in an application, for example, we may extract 1 bit from R, and 3 bits from G, 2 bits from B, which results in a initial hash directory of size $2^{(1+3+2)} = 64$.

In traditional extendible hashing, when a disk block overflows, it is split and a new disk block is allocated. The size of the hash directory doubles if no entry is available to accommodate the new disk block. When a disk block overflows in three-dimensional hashing, like traditional extendible hashing, the hash address space increases and the disk block splits. Unlike traditional extendible hashing, in which a disk block can be split along only one dimension, a disk block in three-dimensional extendible hashing can be split along any of the R, G, B dimensions. We split the disk block along the dimension with the highest variance so that the records can distribute as evenly as possible in the two resulting disk blocks. Suppose, in Figure 4.2, that disk block A, whose hash address is 010, overflows and R dimension has the highest variance—then the disk block is split along R dimension by putting all the records whose R values' second leading bit is 1 to a new disk block D. Since there is no more space in the hash directory to accommodate the new disk block, the address space is doubled, and now the hash address become a 4-bit binary number.

In traditional extendible hashing, new hash addresses are formed by appending one more bit to the end of previous addresses, so in the new hash directory, the directory entries from the previous hash directory interleave the new directory entries. This is not good for implementation: in order to expand a hash directory, the contents of previous hash directory must be moved to a new space. The three-dimensional extendible hashing, instead, puts one more bit at the front of previous addresses to form the new hash addresses, so the directory entries from the previous hash directory occupy the first half of the new hash directory, and the new directory entries take the other half of the new hash directory. This makes the implementation of hash directory more efficient: the hash directory can be expanded by simply attaching more space to the end without changing the contents of the previous hash directory. For example, the 4-bit hash address of colour (01111100, 10101000, 00011001) is 1010, where the most significant bit (1) comes from the second leading bit of R value; this colour value is hashed to disk block D. The contents of the first half of the new hash directory are the same as the previous one, and the entry that points to disk block D is in the other half of the new hash directory (See Figure 4.3).

Since the disk block can be split along any one of the three dimensions, we need to record in which dimension the disk block is split. A data structure named mask track is maintained to keep track of the splitting history of the disk blocks. For example, we record the fact that no disk blocks have been split, except that the disk block with the initial hash address 010 is split along R dimension in the mask track shown in Figure 4.3. Every entry in the mask track is zero, except that the 010 entry is 100.

Now, further suppose that the disk block D in Figure 4.3 overflows and the highest variant dimension within this disk block is B dimension—so that disk block D is split along B dimension by moving all the records whose B values start with 01 to a new disk block E. The hash address space is doubled again. The disk block E is pointed by the entry 11010. The 5-bit hash address of colour (01111100, 10101000, 00011001) is 01010, which is

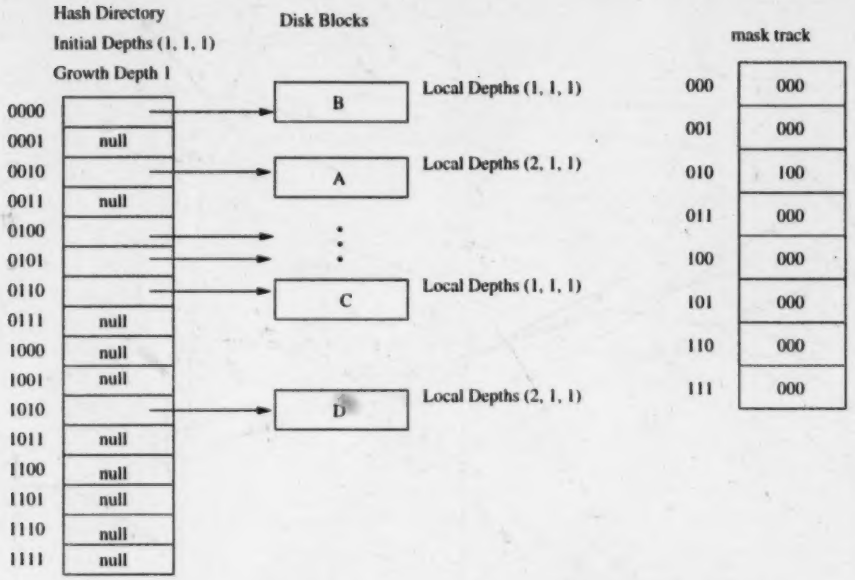


Figure 4.3: Expanded hash directory of the three-dimensional hashing

computed by putting the second leading bit (0) of B value in front of its previous 4-bit hash address 1010. The second level of mask track is created and the value 001 is stored in entry 1010, which means the disk block whose hash address is 1010 is split along B dimension. Figure 4.4 illustrates this situation.

Suppose now that the disk block C in Figure 4.4 overflows and is split along G dimension. A new disk block F is allocated, and all the records whose G values start with 11 are moved from disk block C to F. This time we do not expand the address space. Instead, we simply let the 01110 entry, which previously held a null value, point to disk block F. The contents of entry 110 of the first level mask track are changed from 000 to 010 to record the splitting. If disk block C overflows again, and is split along R dimension this time, a new disk block G is allocated and pointed by the directory entry 10110. The 0110 entry of the second level mask track is changed from 000 to 100. See Figure 4.5.

In general, when a disk block overflows and needs to be split, its local depths are compared to the directory depths. If $(p_1 + p_2 + p_3) = (d_1 + d_2 + d_3 + d_g)$, then there is no address space to hold the new disk block, the directory must be doubled, and d_g is increased by 1. Otherwise, the address space remains unchanged. The original disk block has the same hash address (h) as it had before; the hash address of the new disk block is $2^{(p_1+p_2+p_3)} + h$. The disk block is split along the highest variant dimension. Suppose the R dimension has the highest variance. The disk block is split by moving all records whose R values' $(p_1 + 1)$th leading bit is 1 to the new disk block. The local depths of the two split disk blocks are now $(p_1 + 1, p_2, p_3)$. The entry h of the $(p_1 + p_2 + p_3 - d_1 - d_2 - d_3 + 1)$th

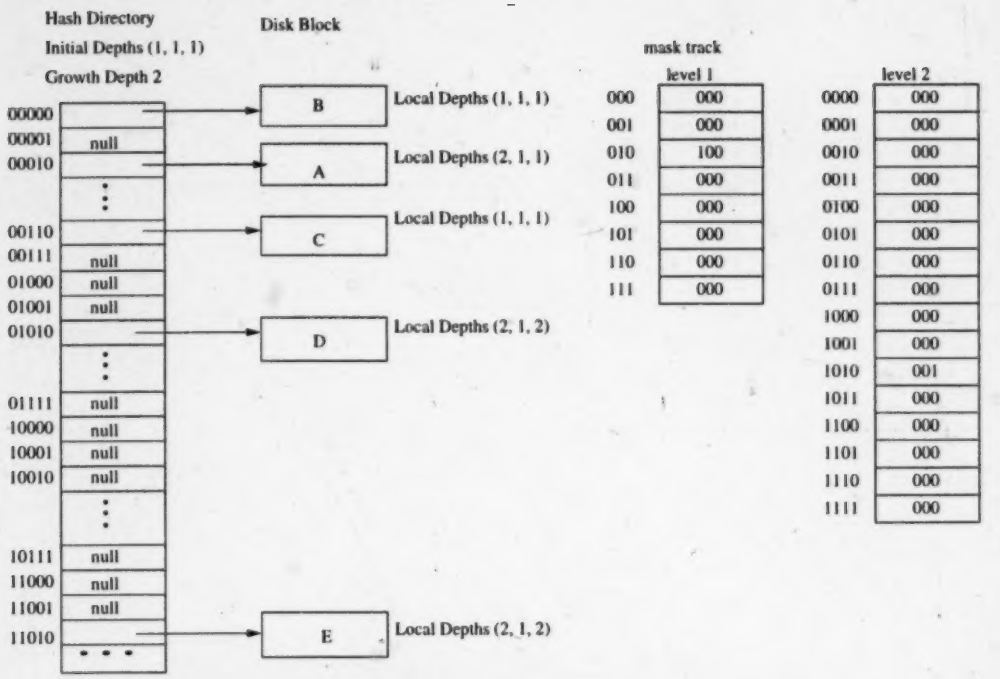


Figure 4.4: Expanded hash directory of the three-dimensional hashing

level mask track is updated to indicate the splitting.

With the above hash structure, a given colour can be mapped to its hash address by the following procedure:

1. Use d_1, d_2, d_3 leading bits from R, G, B values, respectively, and concatenate these bits to form an initial address.

2. Refer to the entry corresponding to the initial address in the first level of mask track. The value in that entry indicates how the disk block represented by this initial address has been split. If the disk block hasn't been split, go to step 5; otherwise the next bit from the split dimension is put at the front of the initial hash address to form a new address.

3. Look up the next level of mask track with the current obtained address. If the mask track entry indicates the disk block do not split, go to step 5.

4. Otherwise, the next bit from the split dimension is put at the front of the current address to obtain a new address, and then go to step 3.

5. Put appropriate number of zeros at the front of the obtained address to make it a $(d_1 + d_2 + d_3 + d_g)$-bit hash address.

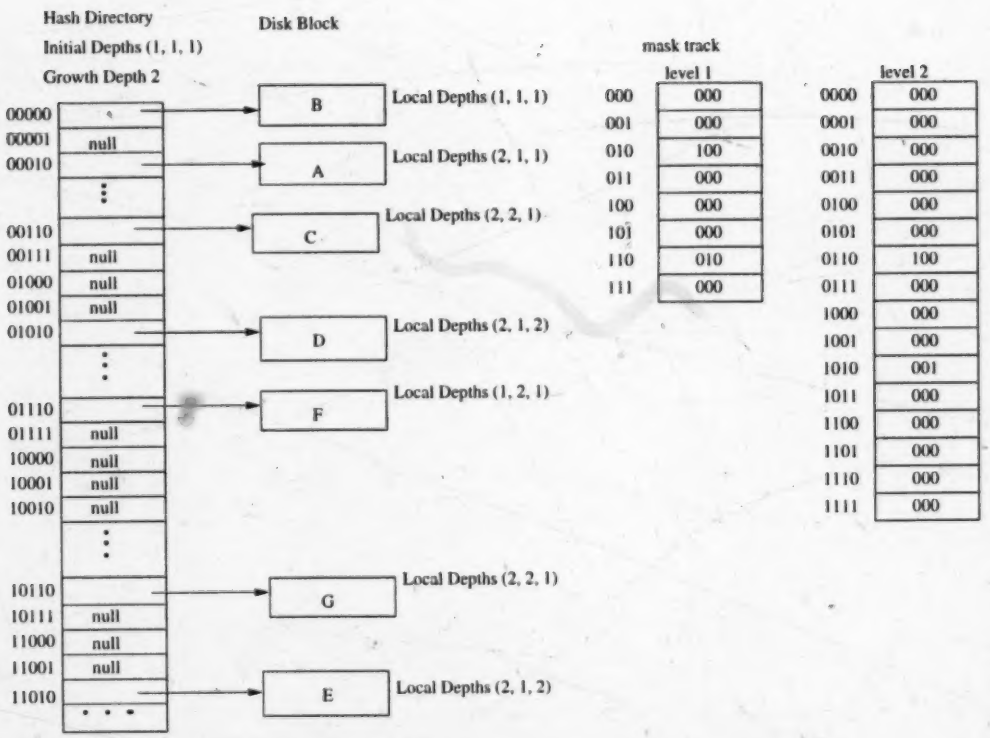


Figure 4.5: Expanded hash directory of the three-dimensional hashing

When records are deleted, the three-dimensional extendible hashing is able to shrink the size of the file system by merging buckets and, when possible, decreasing the size of the directory. Two buckets can be merged only if they are *buddy* buckets, which is to say that they are a pair of buckets resulting from a split. The resulting disk block is pointed by the one with the lower hash address between the two entries which previously pointed to the two buddy disk blocks; the other directory entry will then hold a null value. The content of the corresponding entry in mask track is changed to 000. When the entries of the higher half of the hash directory all have null values, the directory can be shrunk. For example, in Figure 4.5, if disk blocks D and E, and disk blocks C and G are merged, the directory can then be shrunk (Figure 4.6).

To avoid merged disk blocks soon being split again, a merge-threshold [30] is used. The merge-threshold is the percent-occupancy, which the resulting disk block should not exceed when two disk blocks are merged. If the percent-occupancy of resulting disk block is above the threshold, merging should not occur.

4.2 Multi-dimensional extendible hashing

Three-dimensional extendible hashing can easily be generalized to handle n-dimensional data. The directory of multi-dimensional extendible hashing has n initial depths (d_1, d_2, ..., d_n) and a growth depth d_p. The disk block has n local depths (p_1, p_2, ..., p_n). When a disk block overflows, it can be split along any of the n dimensions. We choose to split the disk block along the dimension which has the highest variance. If the disk block is split along the i-th dimension, the local depths of the resulting two disk blocks are (p_1, p_2, ..., $p_i + 1$, ..., p_n). The directory is doubled when necessary. The mask track is still used to keep track of the disk block splitting history.

4.3 Range Query Algorithm

Given a query colour value (a, b, c) and a distance threshold ϵ where the distance refers to the Euclidean (L_2) distance, a range query is to find all the colour values residing in the search region, a sphere whose center is (a, b, c) and radius is ϵ.

The initial hash directory determines the initial partition of the colour space. For example, the hash directory in Figure 4.2 determines the partitions shown in Figure 4.7. Each disk block pointed at by the initial hash directory represents one of the partitions. During the growth of the hash directory, the disk blocks may have been split. The regions represented by the resulting disk blocks are further partitions of the original disk block region. The range query algorithm will first locate all the disk blocks whose regions overlap the search region, and then examine the contents of the disk blocks. Since the shape of the disk block region is a rectangular solid, and it is easier to decide whether two rectangular solids overlap, a minimum bounding cube of the sphere, instead of the sphere itself, is used

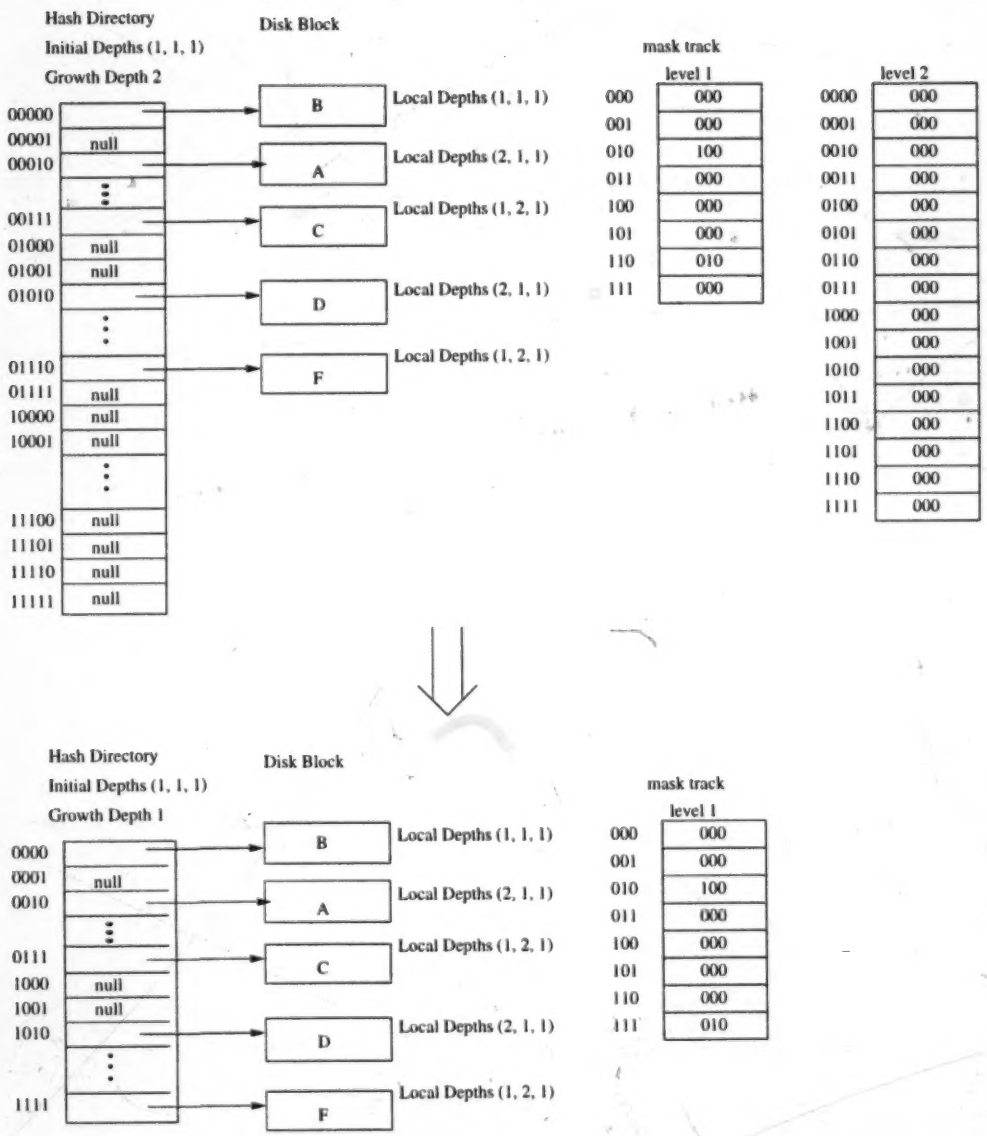


Figure 4.6: Merging disk blocks and shrinking directory

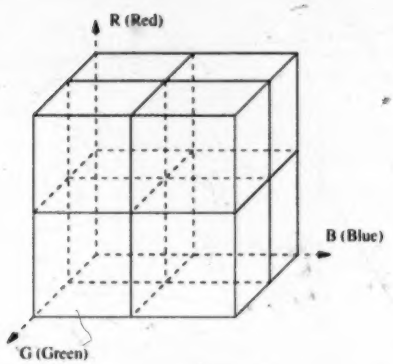


Figure 4.7: Initial partitions of the colour space

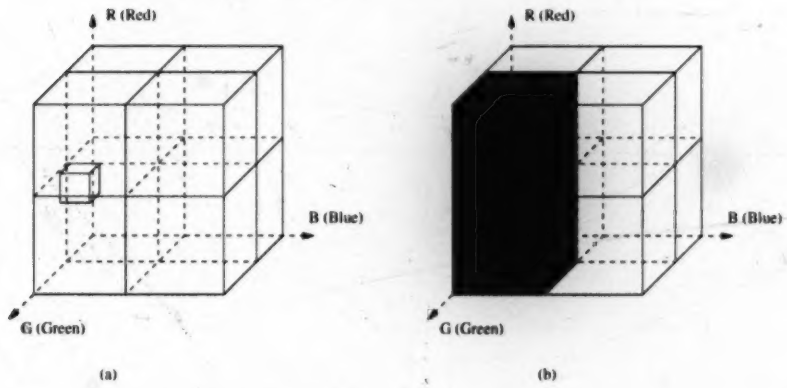


Figure 4.8: First step of the search algorithm

during the procedure of locating the disk blocks.

The first step of this algorithm is to compute the set of initial hash addresses of the partitions which overlap the cube. For example, given a colour value (R, G, B) representing in binary numbers (01111100, 10101000, 00011001) and a distance threshold of 12, the minimum bounding cube of the search region is ([01110000, 10001000], [10011100, 10110100], [00001101, 00100101]. Thus, for the hash directory whose initial depth is (1, 1, 1) as shown in Figure 4.2, the set of initial hash addresses whose corresponding regions overlap the search region is { 010, 110 }. This is illustrated in Figure 4.8. The small cube in Figure 4.8 (a) is the minimum bounding cube of the search region. The two blue cubes in Figure 4.8 (b) is the initial partitions that overlap the minimum bounding cube.

The mask track is looked up level by level to find out how these initial partitions are further partitioned. For example, the first level mask track in Figure 4.5 indicates the 010 disk block has been split along R dimension, resulting in two disk blocks, 0010 and 1010. Only the 1010 disk block still overlaps the minimum bounding cube; the 0010 disk block is outside the cube. This situation is shown in Figure 4.9. The lower blue cube is cut into

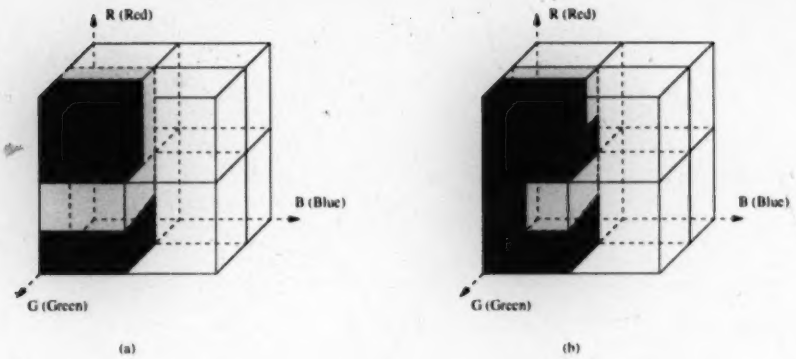


Figure 4.9: Further partitioning of initial partitions

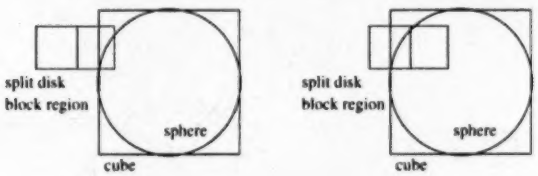


Figure 4.10: Split disk blocks

two halves. Only the upper half still overlaps the minimum bounding cube. When we look up the 110 entry in the first level mask track, we know that this disk block has been split along G dimension. Only the 0110 disk block still overlaps the cube. This is also illustrated in Figure 4.9. The upper blue cube is cut into two halves, and only the yellow half still overlaps the minimum bounding cube. So the set of addresses that is in consideration becomes { 1010, 0110 }, where the 010 is replaced by 1010, and 110 is replaced by 0110. The addresses 0010 and 1110 are discarded, because the regions represented by them are outside the bounding cube.

The second level mask track tells us that the 1010 disk block has been split along B dimension, and the 0110 disk block along R dimension. The regions represented by the addresses 11010 and 10110 are outside the bounding cube, so now the address set becomes { 01010, 00110 }. Figure 4.9 (b) illustrates this situation. The yellow rectangular solids are divided into halves, and the red parts still overlap the minimum bounding cube.

In general, when a disk block overlapping the bounding cube is split, there are two possibilities: either only one resulting disk block overlaps the cube, or both of them do, as shown by Figure 4.10. Only the disk blocks overlapping the cube need to be further checked.

Keep looking up the next level mask track until all the addresses in the set are final; that is, the corresponding disk blocks do not split. According to Figure 4.5, the address set { 01010, 00110 } is final. Then further check the relative position between the disk block and the sphere. There are three possibilities, as shown by Figure 4.11. If the disk block

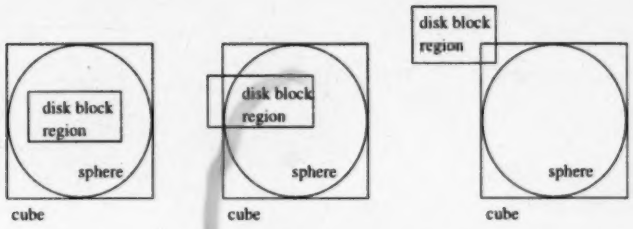


Figure 4.11: Non-split disk blocks

is contained in the sphere, all the colour values in this disk block are returned as query results; if the disk block intersects with the sphere, the colour values need to be examined one by one, and those qualifying colour values are returned; if the disk block is outside the sphere, it is discarded.

The algorithm illustrated by the above example is summarized as follows. The input of this algorithm is a range query: colour value (a, b, c) and a distance threshold ϵ. The output is the set of qualifying data records. The search region, i.e., the sphere whose center is (a, b, c) radius is ϵ, is referred to as $SPHERE$. The minimum bounding cube of $SPHERE$ is referred to as $CUBE$.

1. Set up a set S which contains all the initial hash addresses whose corresponding regions overlap $CUBE$. Set up an empty set R to store the query result.

2. If set S is empty, output R; otherwise remove an address a from S. If a is a final hash address, go to step 4; otherwise go to the next step.

3. Look up the mask track to find out how the region represented by address a is split. Consider the two resulting regions, and add the addresses of the resulting regions that overlap $CUBE$ to set S. Go to step 2.

4. If the region represented by address a overlaps $SPHERE$, go to the next step; if the region is contained in $SPHERE$, add all the records in the corresponding disk block to set R, and then go to step 2; if the region is outside the sphere, go to step 2 directly.

5. Examine the data records in the disk block one by one and add those qualifying records to set R. After examining all the data records, go to step 2.

4.4 Multi-dimensional Linear Hashing

In multi-dimensional extendible hashing, while the disk blocks can be split along any of the multiple dimensions, the directory only grows in one dimension. It is the mask track that maps multiple dimensions to one dimension. The mask track is a crucial structure which enables extendible hashing to handle multi-dimensional data. Applying the mask

track to other hashing algorithms, such as linear hashing, can also enable them to handle multi-dimensional data.

Linear hashing, first introduced in [27], is similar to extendible hashing in its use of more bits of a hash address as the address space grows. The directory of extendible hashing grows by doubling itself, which makes the growth of the directory very fast, especially when the data distribution is skewed, while linear hashing allows the address space to grow linearly. Multi-dimensional linear hashing is built by adding the mask track structure to the traditional linear hashing.

Linear hashing does not split the disk block that overflows. Instead, it splits the disk blocks in the order that they are placed in the hash address space. The following example will provide more details. The example begins with an address space of four (2-bit hash address). See Figure 4.12(a).

Suppose that as a data record is inserted, disk block b overflows, which forces a split, but instead of splitting disk block b, we split disk block a (Figure 4.12(b)). The reason for this is that we are extending the address space linearly, and disk block a is the next one that must split to create the next linear extension. The new allocated disk block is A. Now the hash addresses of disk block A and a are 3-bit. Since the data are in an n-dimensional space, disk block a can be split along any of the n dimensions. However, only the dimension which has the highest variance is chosen. Suppose that the i-th dimension has the highest variance. The records in disk block a are divided between disk block a and A according to the i-th dimension data. The mask track records the fact that disk block a is split along the i-th dimension. Since disk block b was not the one that we split, the overflowing record is placed into an overflow disk block w.

When disk block d overflows, disk block b is the next one to split and extend the address space. So the records in disk block b and w are distributed between b and the new disk block B. The record overflowing disk block d is placed in disk block x. Assume the split occurs along the j-th dimension. A corresponding entry is created in the mask track to record this split (Figure 4.12(c)).

Figure 4.12(d) shows what happens when disk block x overflows. Disk block c is next in the extension sequence, so a new disk block C is allocated. A new overflow disk block y is chained to disk block x to accommodate the overflowing record. Assume that disk block c is split along the k-th dimension, which is recorded in the mask track.

Next assume that disk block B overflows. The overflow record is placed in disk block z. The overflow also triggers the extension to disk block D, dividing the records of disk blocks d, x, and y between disk blocks d and D. The dimension l along which these disk blocks is split is recorded in the mask track.

At this point all of the disk blocks use 3-bit hash addresses, and an expansion cycle is finished. The pointer for the next disk block to be split returns to disk block a to get ready for a new cycle, that will use 4-bit hash addresses to reach new disk blocks. If disk block A

overflows, which results in the split of disk block a, the second level of mask track must be created in order to record the dimension along which disk block a is split. In this case, for example, m is stored in the 000 entry of the second level mask track. See Figure 4.12(f).

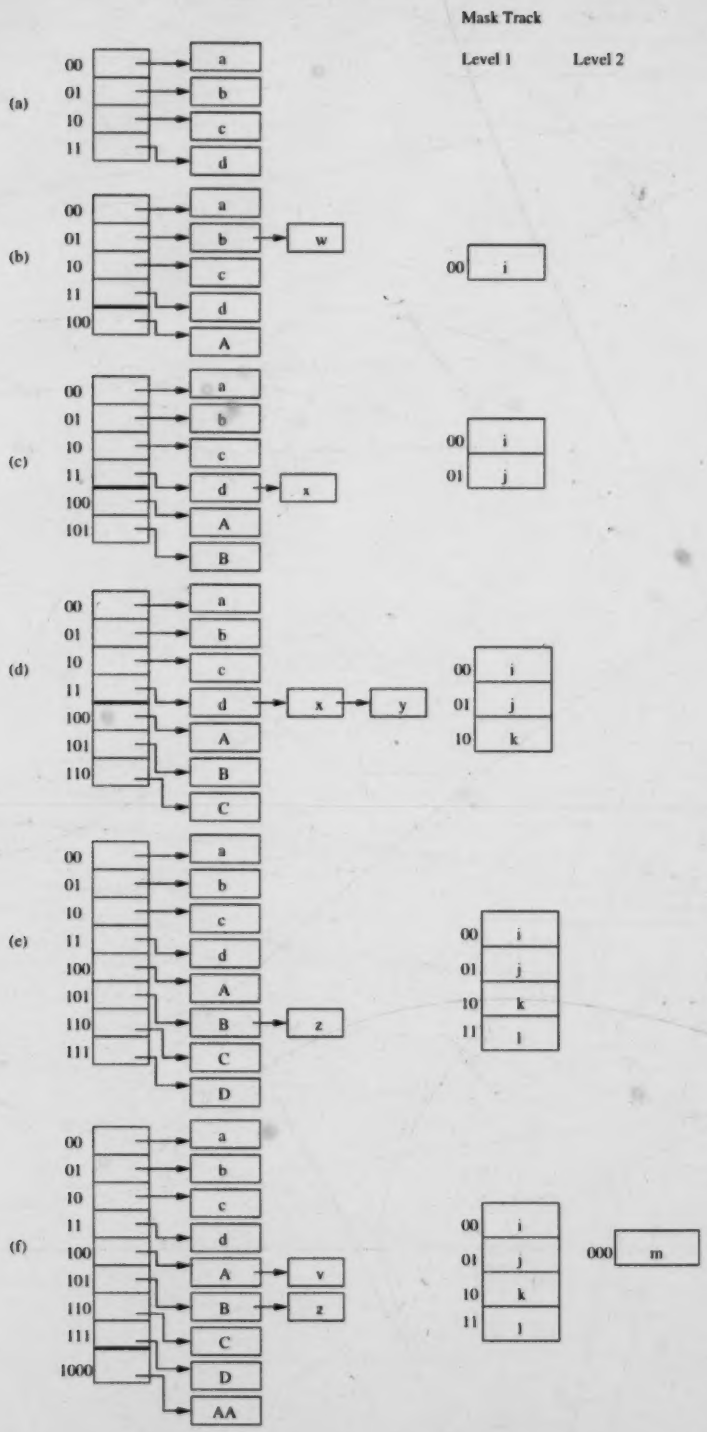


Figure 4.12: The growth of address space in multi-dimensional linear hashing

Chapter 5

Experiments

In the context of indexing images for similarity searches, query performance is much more important than update performance because image databases will typically rarely be updated, and often the updating will be done as an off-line processing step. In contrast, query performance must be as fast as possible since similarity queries will typically be performed online, interactively, and will account for most accesses to the indexing structure. For this reason, the experiments described here focus on query performance. Three-dimensional extendible hashing is compared to the SR-tree (SR-tree was introduced in Section 2.1.3). The SR-tree is chosen because it is one of the best multi-dimensional index structures. According to [23], SR-tree outperforms SS-tree for various dimensionality from 1 to 64, which means although SR-tree is designed for high-dimensional data, it is also effective for lower dimensionality. Disk access is one of the most time-consuming operations, but it alone does not adequately measure performance because the computation needed to compare two image features is complex, and also costs time. Because of this, we compare both I/O performance and CPU time performance.

Although, as a general rule, space should be traded for performance, the space utilization should still be reasonable. So we also examine the space utilization of three-dimensional hashing.

The precision and recall[1] are not studied. The experiments concentrate on testing the performance of three-dimensional structure, not evaluating the effectiveness of colour histogram comparison metrics.

5.1 Experiment Setup

We first investigate the average colour distribution of real images so that we can generate synthetic data according to the real data distribution. The synthetic data are generated to form a large dataset so that we can test the scalability of the index structures.

[1] Recall is a measure of how well an information search and retrieval system finds ALL relevant documents on a searched for topic, even to the extent that it includes some irrelevant documents. Precision is a measure of how well such a system finds ONLY relevant documents on a searched for topics, even to the extent that it skips irrelevant documents.

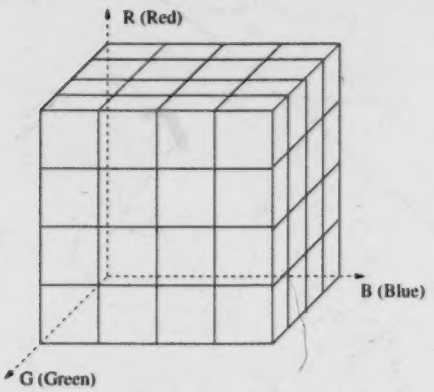


Figure 5.1: Initial colour space partition

An RGB colour model is used in the experiments. The colour space is quantized in order to compute colour histograms. All possible 256^3 colour values in the RGB colour model are grouped into 64 values, i.e., each of the R, G, B dimensions is quantized into 4 equal length intervals, which results in 64-bin colour histograms. Average colours are then computed from colour histograms [21].

The test image database consists of 6601 images from a published photo collection CD ROM [1]. These images fall into six categories, including 919 animal images, 1087 people images, 1161 plant images, 1510 scenic images, 963 structure images, and 961 transportation images. The sizes of these images vary, but they are all around 760×760. A colour histogram and an average colour is computed for each of the images. The average colours (three-dimensional points) constitute the experiment dataset.

The number of entries in the initial hash directory is set to 64, i.e., the initial hash address has 6 bits, with 2 bits from each of the R, G, B components. This is equivalent to dividing the colour space into 64 partitions, as shown by Figure 5.1. How the image average colours distribute in the colour space, particularly in the 64 partitions, is of interest. Figure 5.2 shows the distributions, and we notice that the six distributions are very similar. Most of the average colours are in the 21st partition (the partition number starts from 0), which corresponds to the colours with their R, G, B values in the range of $[64, 127]$. Because of this similarity, there is no point in differentiating images into six categories, and all the images are put in one dataset.

In order to test the scalability of the hash structure, we need much larger datasets. Synthetic data are generated for this purpose. To simulate the real data distribution more accurately, we also examine the distribution of the three colour components which make up the average colour values. Figure 5.3 presents the data distribution of the 6601 real images, including the average R, G, B value distributions, as well as the average colour value distribution. The synthetic data are generated in two steps. First R, G, B values

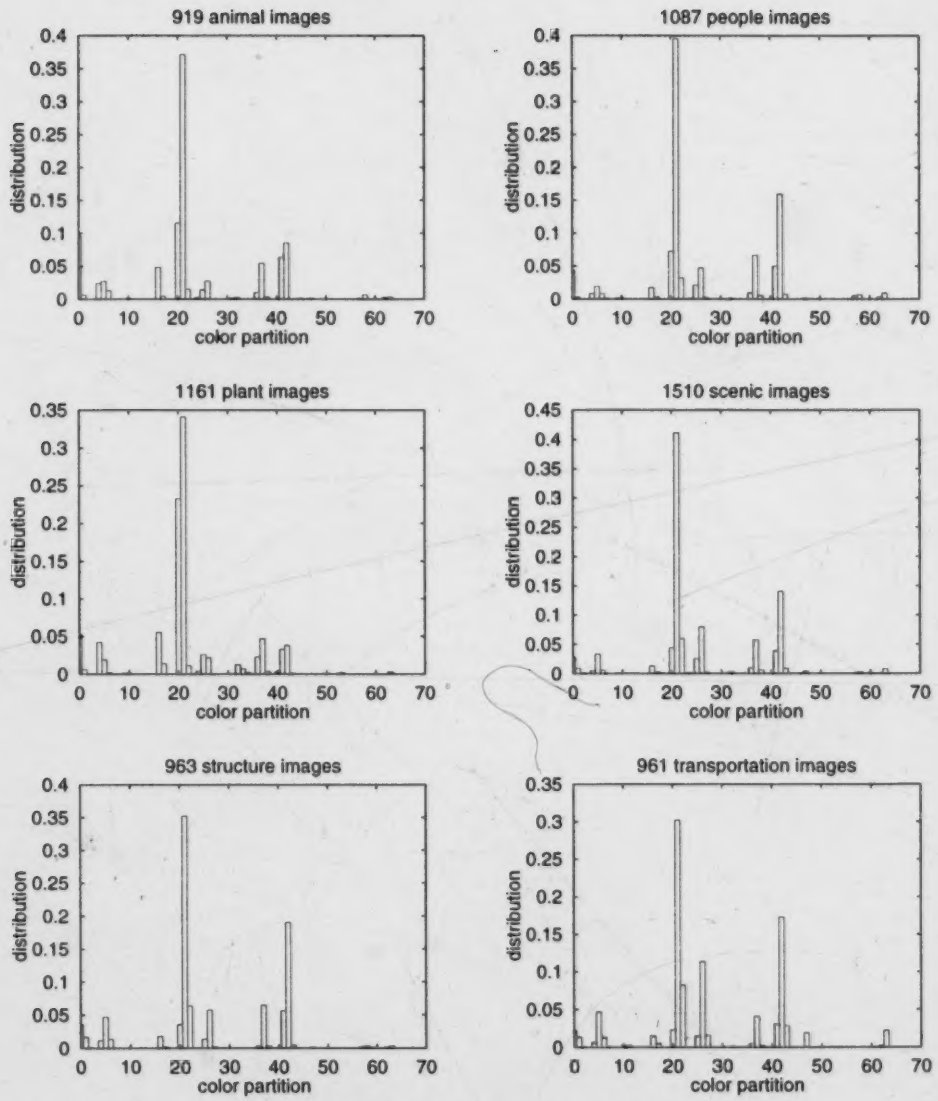


Figure 5.2: Average colour distributions

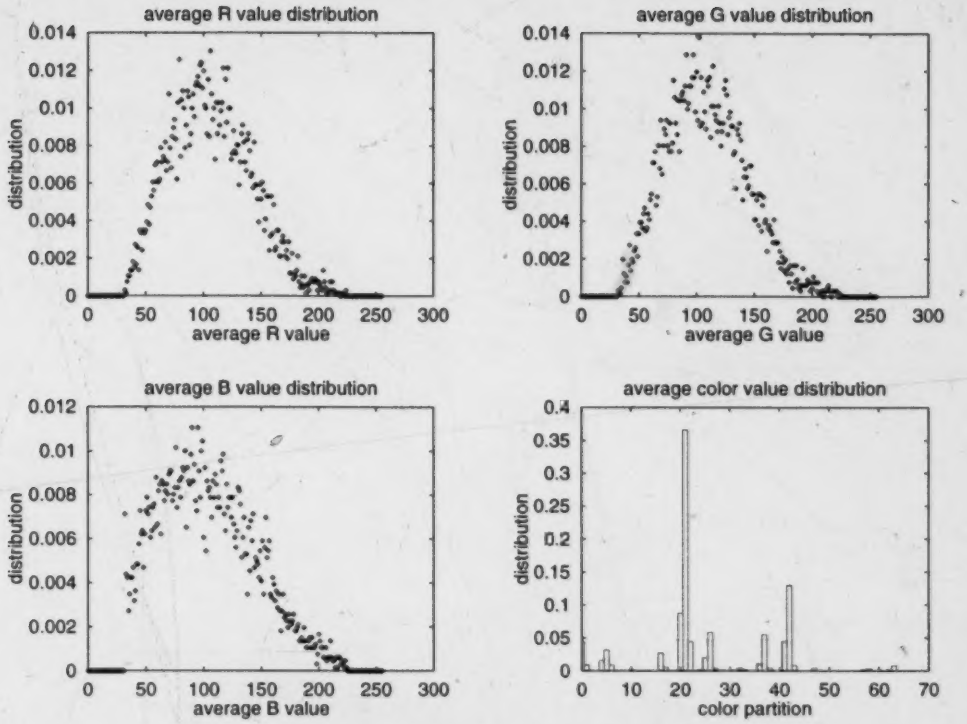


Figure 5.3: Real data (6601 images) distribution

are generated separately according to the distributions shown in Figure 5.3. Then these R, G, B values are combined randomly to make up colour values with the condition that the colour values are distributed in the 64 colour partitions in the same way as shown in Figure 5.3.

We generate six datasets of the size 5,000, 10,000, 50,000, 100,000, 500,000, and 1,000,000 points for the experiments. The 5000-size dataset contains only real data. All the other datasets are semi-real, i.e., they contain the 6601 data points (average colour values) computed from the 6601 real images, and the rest of the data are synthetic.

We implement the hashing algorithm in C++. The SR-tree code is provided by its authors (downloaded from http://www.rd.nacsis.ac.jp/~katayama/homepage/research/srtree/). The disk block size is set to 4096. A hash bucket or a SR-tree leaf node can hold a maximum of 511 data points. The fan-out of the SR-tree is 88. These experiments were run on a Sun Ultra sparc 10 workstation.

	Dataset Size	5000	10000	50000	100000	500000	1000000
SR-tree	Height	2	2	3	3	3	3
	Non-Leaf Nodes	1	1	3	4	17	35
	Leaf Nodes	11	24	113	230	1143	2319
Hash	Disk Blocks	48	60	171	309	1440	2867

Table 5.1: Disk blocks used by SR-tree and Hash

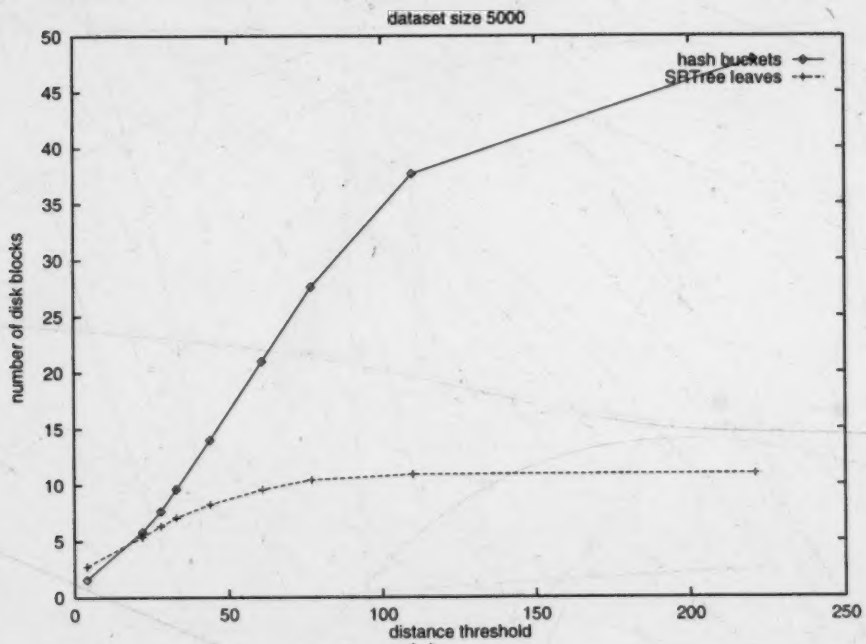
5.2 Experiment Results and Discussion

5.2.1 Query Performance

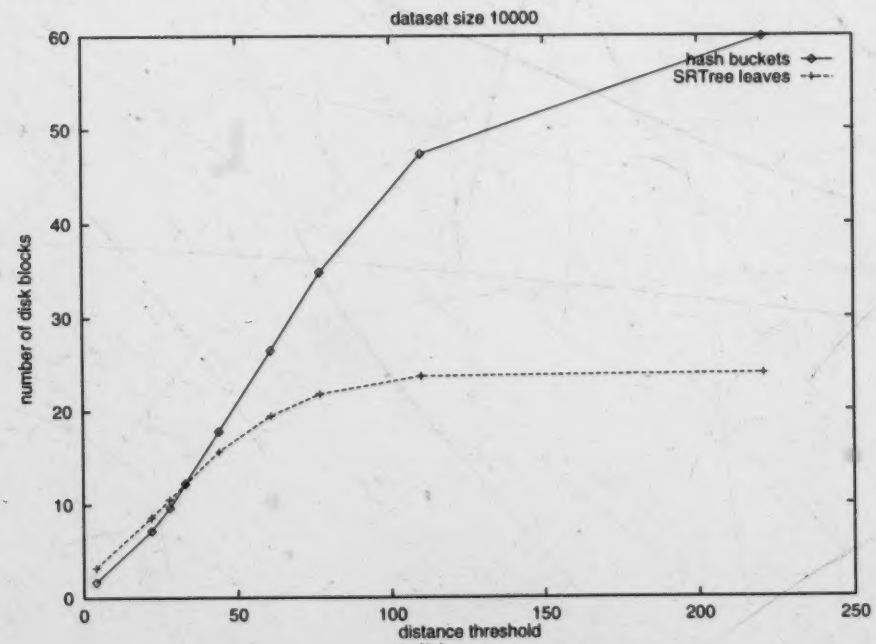
We run range queries (see Section 4.3 for the definition of range queries) on the hash structure and the SR-tree to compare their query performance. 500 query points are randomly generated with the same data distribution as the dataset. Queries are run for nine distance thresholds: 4, 22, 28, 33, 44, 61, 77, 110, 221, which are approximately 1%, 5%, 6.25%, 7.5%, 10%, 13.75%, 17.5%, 25%, 50% of the maximum Euclidean distance between two colour values. (The maximum distance is the distance between colour (0, 0, 0) and colour (255, 255, 255), which is $\sqrt{3 * 255^2} = 441.67296$.) The six datasets mentioned above are used in the experiments. We run 500 queries for each of the distance thresholds and datasets, and then compute the average from the obtained 500 values.

Figure 5.4 (a)–(f) presents the experiment results with regard to I/O performance. The horizontal axis denotes the distance threshold of the queries, and the vertical axis indicates the average number of disk blocks read in order to process the queries. The hash structure outperforms the SR-tree when the search distance threshold is small; when the search distance threshold is large, however, the SR-tree is better.

The reason for this is related to the way these two index structures divide the colour space: the hash structure divides the space into non-overlapping partitions, while the SR-tree divides the space into overlapping portions. This requires that more disk blocks be read when searching the SR-tree because if the search region intersects with the overlapping region of two disk blocks, then these two disk blocks have to be read. On the other hand, the hash structure divides spaces finer than the SR-tree: as shown by Table 5.1, the number of disk blocks in the hash structure is larger than the number of leaves in the SR-tree. This forces more disk blocks to be read when searching on the hash structure. When the distance threshold is small, a finer partition does not affect performance much because the region that needs to be searched is small anyway. In this case, overlapping becomes the dominant factor, which makes the hash structure outperform the SR-tree. When the distance threshold is large, overlapping does not matter anymore, because even if the portions were not overlapped, the search region is large enough to include them anyway. On the contrary, the larger the distance threshold, the more the hash structure is penalized for its finer partition. Thus the SR-tree outperforms the hash when the range threshold is large. We also note that the performance crossing point moves to the higher end of the horizontal

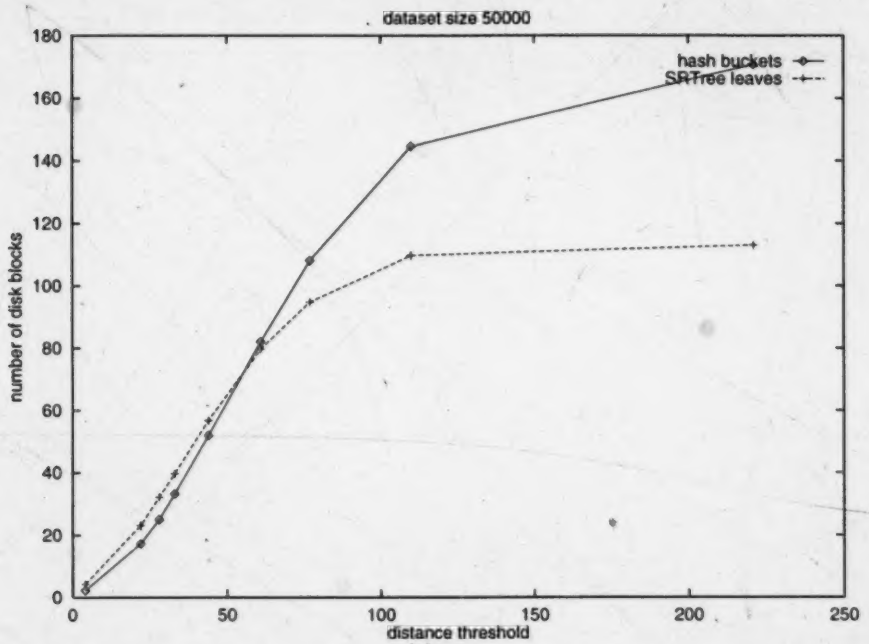


(a)

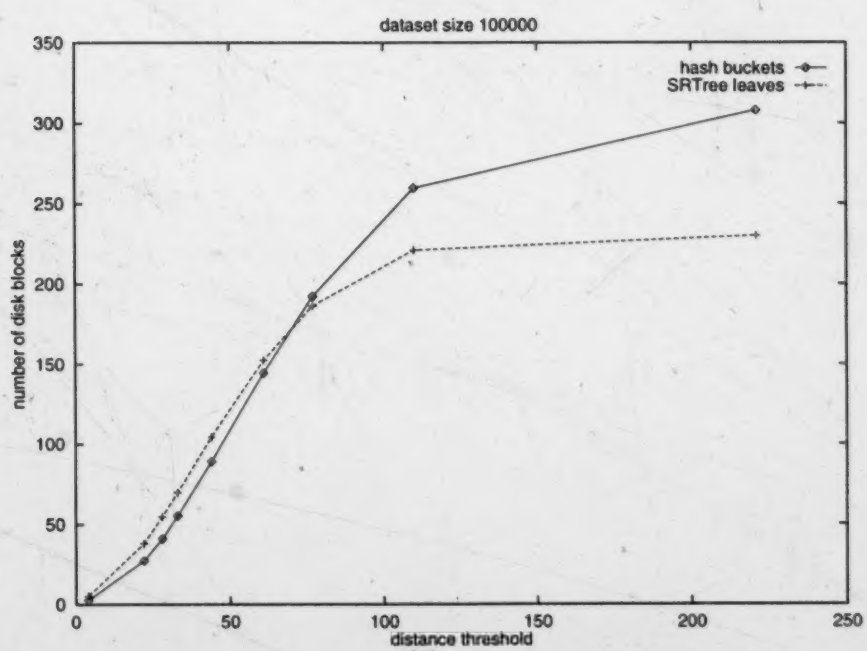


(b)

Figure 5.4: I/O performance of Hash and SR-tree

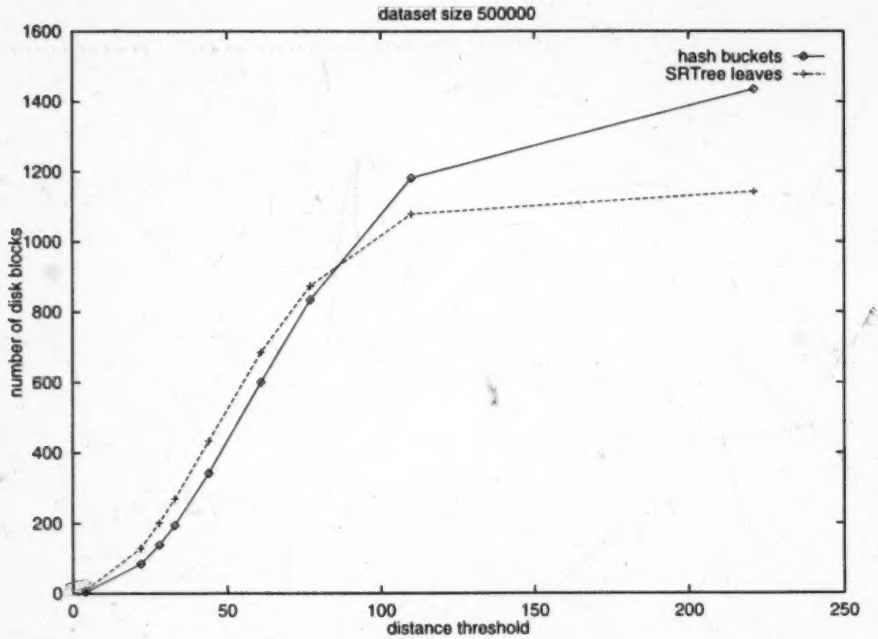


(c)

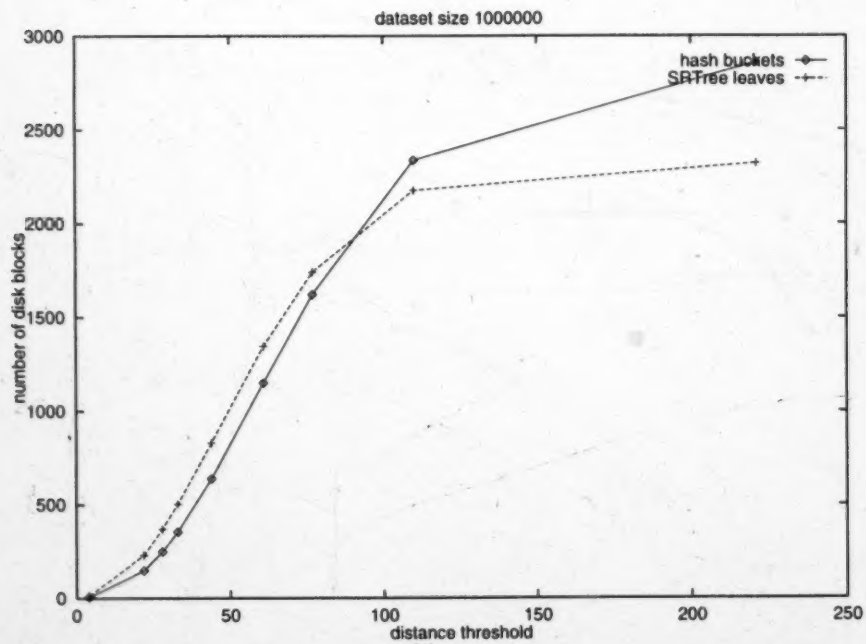


(d)

Figure 5.4: I/O performance of Hash and SR-tree (continued)



(e)



(f)

Figure 5.4: I/O performance of Hash and SR-tree (continued)

axis when the dataset size gets larger, which means that the hash performs better (or the SR-tree works worse) for larger datasets.

The I/O performance of the hash is satisfactory, because in order to retrieve similar images, the distance threshold is usually small.

The CPU time performance of the hash is much better than that of the SR-tree, as shown in Figure 5.5 (a)–(f).

The reason is related to the search algorithms, which, of course, are based on the index structures. For the hash structure, the initial partition of the space is pre-determined, so the initial hash addresses of the disk blocks whose regions overlap the search region can be computed, i.e., no search operations are needed here. For the SR-tree, the root node must be searched first to decide which child node overlaps the search region, and then the child nodes are searched in turn. Searching the nodes is time-consuming because the fan-out of the SR-tree is fairly large.

Another reason for the superior performance of hashing to SR-tree is how the two structures compare the disk block region to the search region. For the hash structure, the search starts from the candidate disk blocks obtained by computation. These disk blocks may have been split during the growth of the hash structure. Region comparisons are needed to determine whether the resulting disk blocks overlap the search region. Only the dimension along which the disk block has been split is compared because the original disk block overlaps the search region. This results in only two possibilities (Figure 4.10): either both of the resulting buckets overlap the search region, or only one of them does. For the SR-tree, in order to determine whether the node region overlaps the search region, comparisons must be done for all the three dimensions, since there are no hints about the relative position between the node region and the search region.

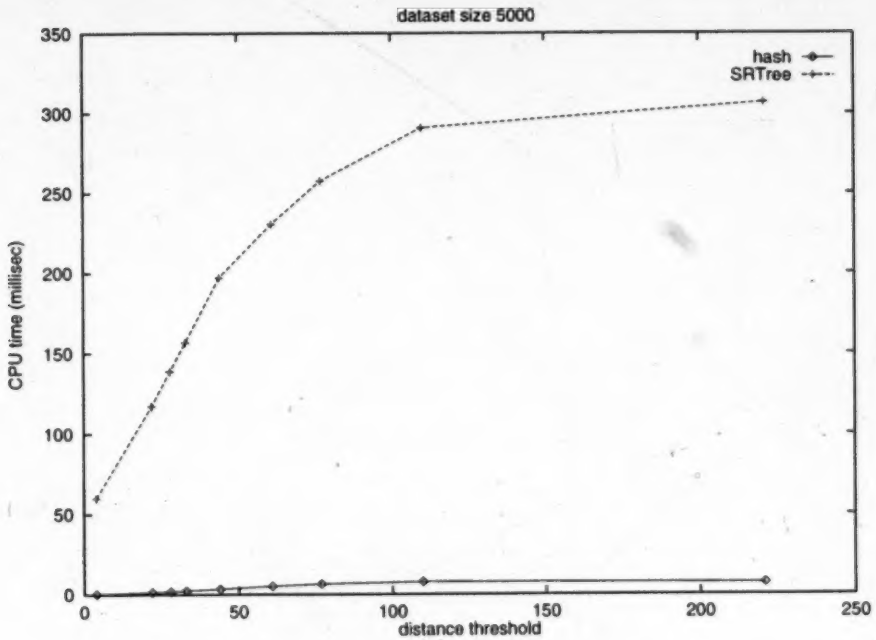
In both Figures 5.4 and 5.5, the curves gradually level off as the distance threshold gets larger. This reflects the fact that these two index structures lose their effectiveness when the distance threshold is large. The search degrades gradually to a sequential scan, i.e., almost every data record in the database will be visited.

Figure 5.6 and 5.7 shows that three-dimensional extendible hashing is scalable with regarding to both I/O performance and CPU time. The horizontal axes denote the dataset size. The vertical axes in Figure 5.6 indicate the average number of disk blocks read. The vertical axes in Figure 5.7 indicate the CPU time spent. Both the number of disk blocks and the CPU time grow linearly as the dataset increases in size.

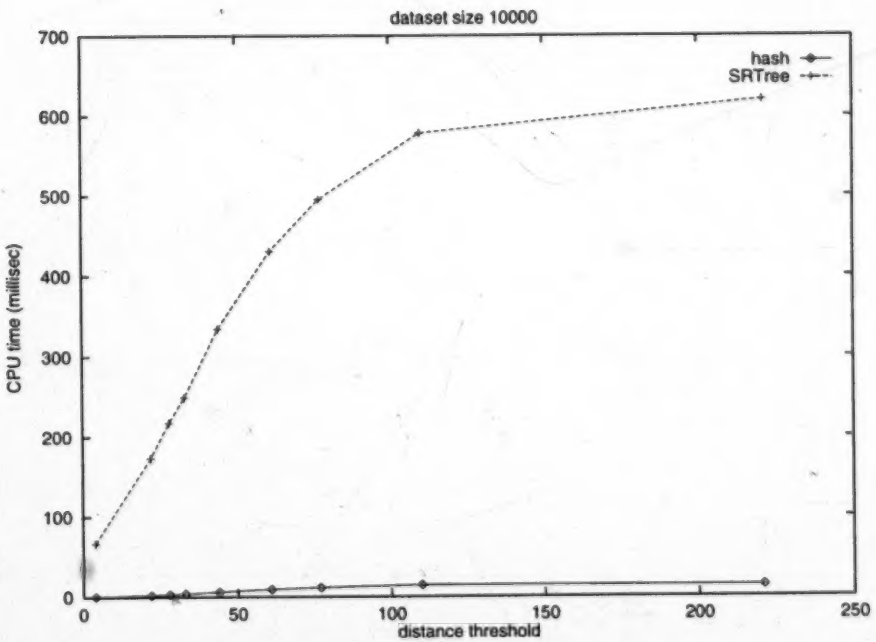
5.2.2 Space Utilization

We also run experiments to examine the space utilization of the hash structure. We are interested in two scenarios: the continuously growing database (repeated insertions) and the steady-state database (as many insertions as deletions).

Figure 5.8 presents the experimental results for the growing database. We measure

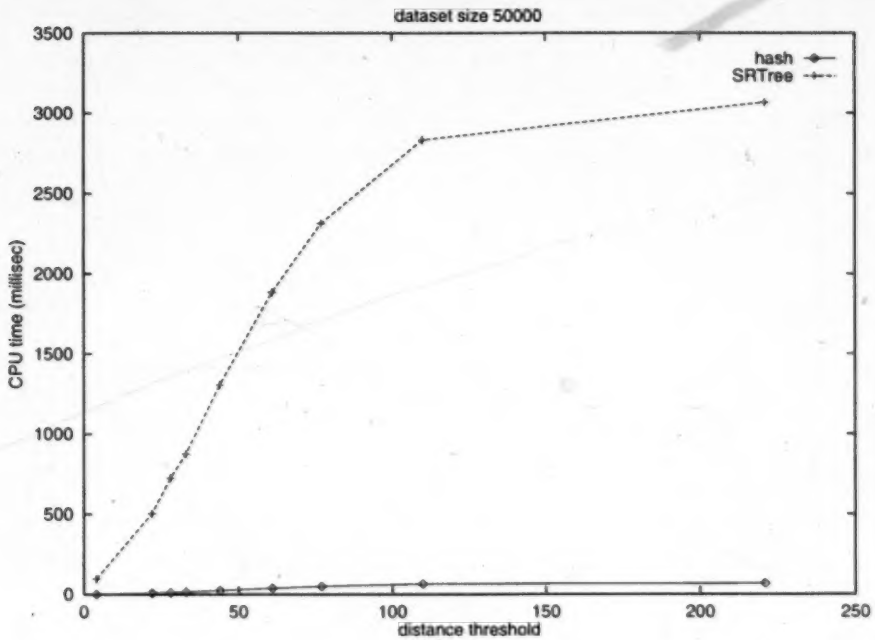


(a)

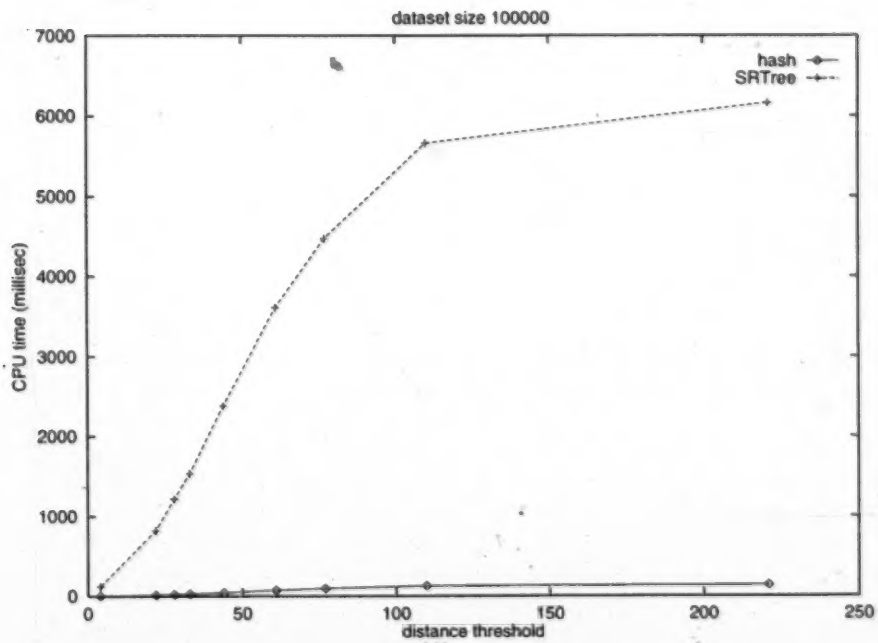


(b)

Figure 5.5: CPU time performance of Hash and SR-tree

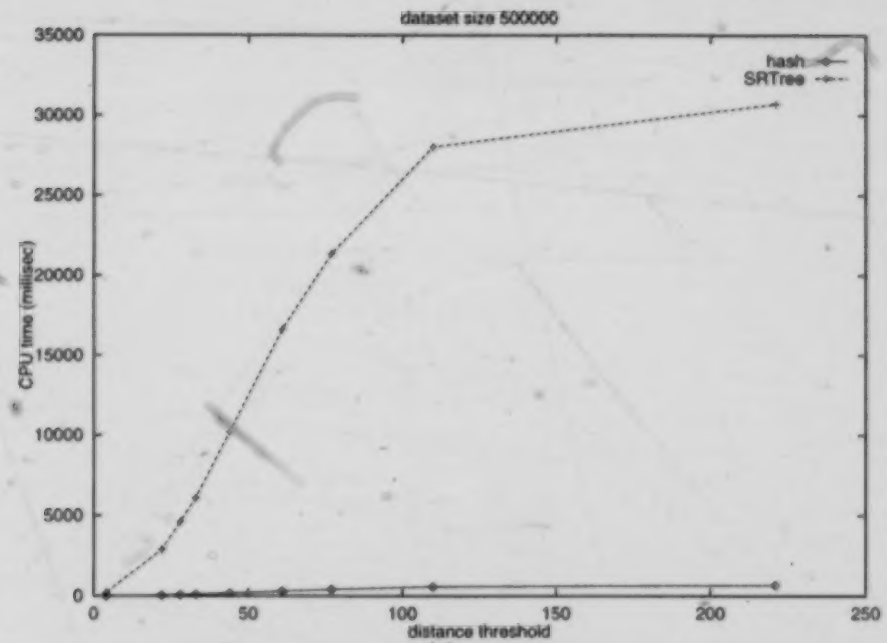


(c)

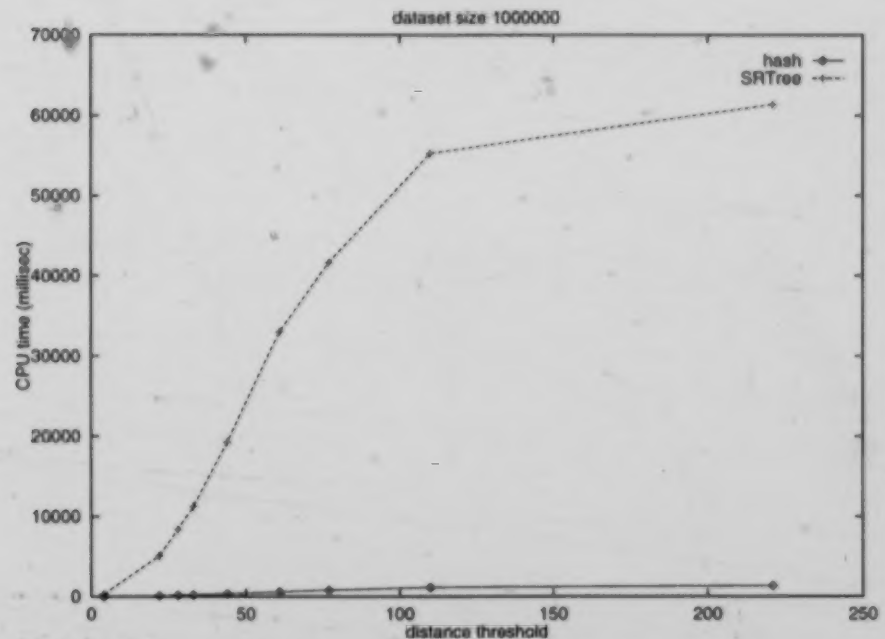


(d)

Figure 5.5: CPU time performance of Hash and SR-tree (continued)



(e)



(f)

Figure 5.5: CPU time performance of Hash and SR-tree (continued)

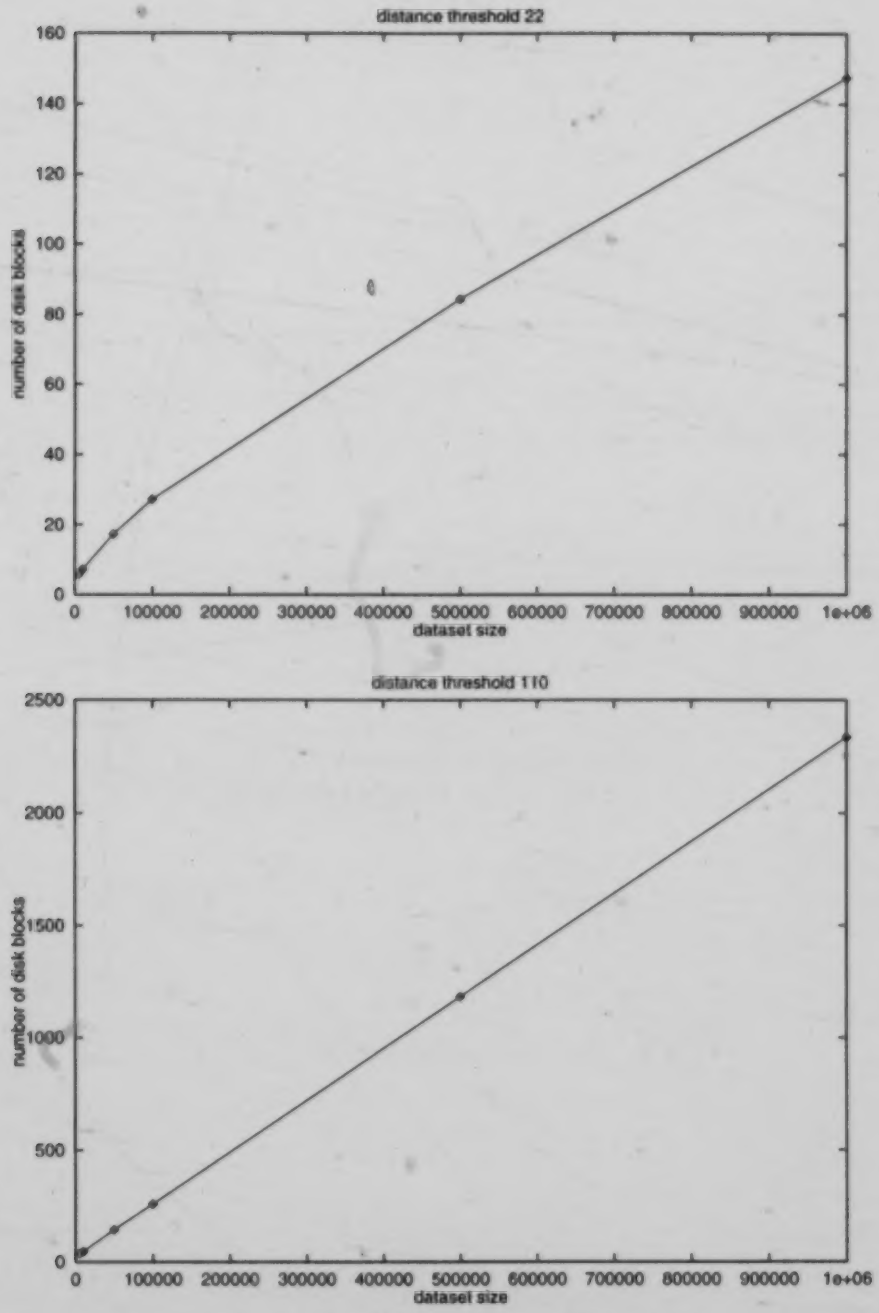


Figure 5.6: Scalability of Hash regarding I/O performance

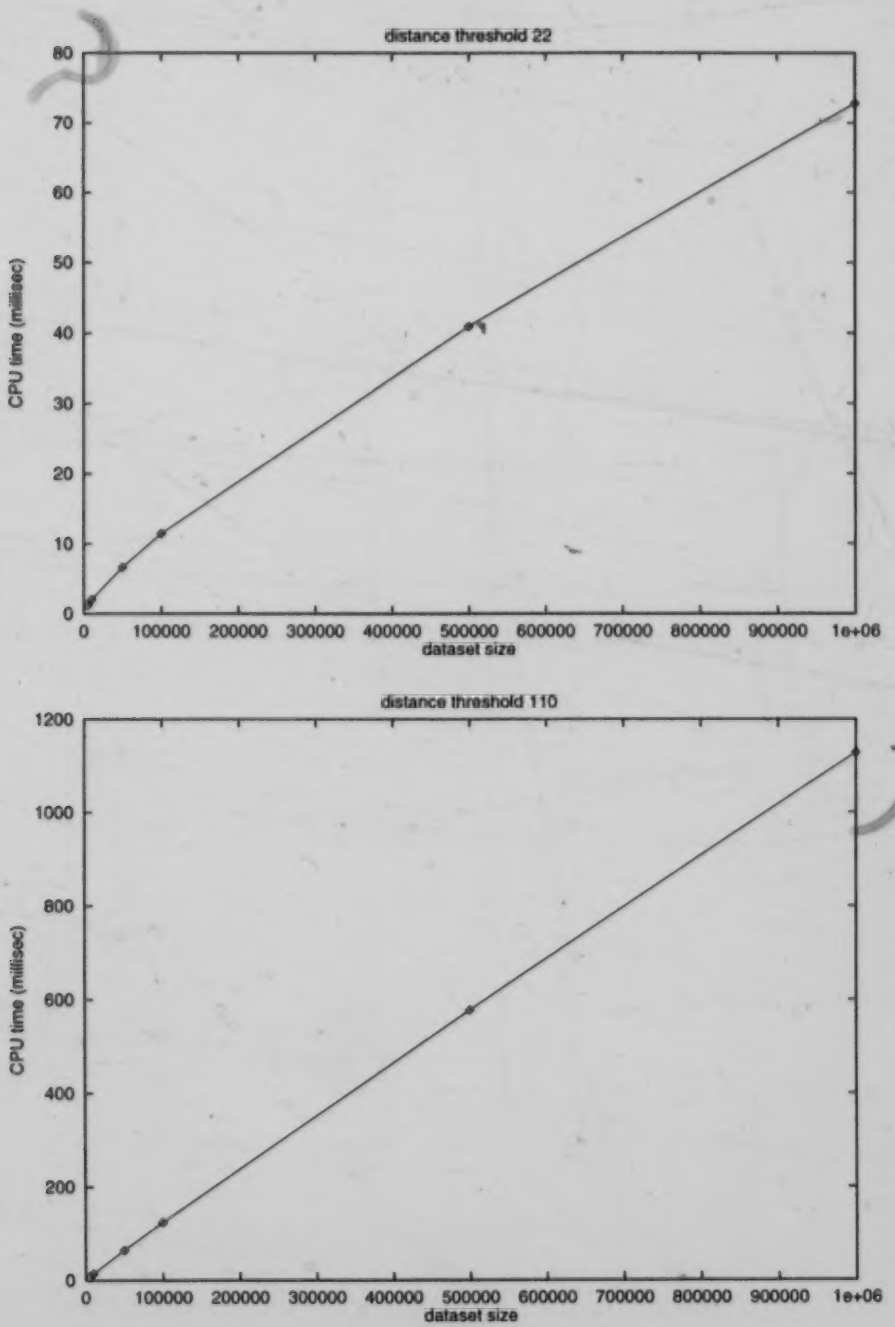
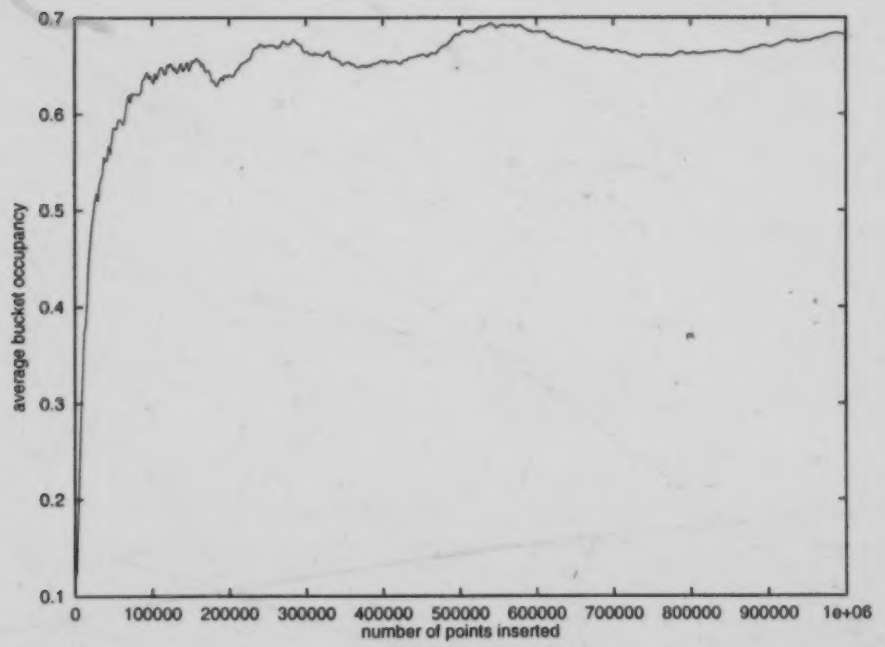


Figure 5.7: Scalability of Hash regarding CPU time

average disk block capacity every 2000 insertions. The average disk block capacity shows a steady state behavior with small fluctuations of around 66 percent, when the number of inserted points is large enough (larger than 100,000). The directory entry utilization curve is zig-zag because the directory grows by doubling itself. Once the directory doubles, the utilization drops dramatically. It then grows gradually until the next time the directory doubles. The directory entry utilization is poor because the data distribution is very skewed and the directory size keeps doubling when the dataset size grows.

For the experiments with steady-state dataset, the hash structure is first initialized with 500,000 points, then 250,000 randomly generated points are inserted into, and 250,000 random points are deleted from, the hash structure. Insertions and deletions are interleaved in a random way. The experiments are conducted with different *merge-thresholds*, defined as the percent-occupancy that the resulting disk block should not exceed when two disk blocks are merged. The merge-thresholds that are considered are 100%, 90%, 70% and 50%. Figure 5.9 shows the experimental results. The solid line in Figure 5.9 is for 100% merge-threshold and the dashed line is for 90% merge-threshold. We measure average disk block occupancy every 2000 operations (insertions and deletions). The average disk block occupancy keeps dropping, which indicates that the space partition is less optimized during the deletions and insertions. The hash structure is nondeterministic in partitioning the colour space, i.e., different sequences of insertions will result in different partitions. Space partition which is good at one time may not be good after the dataset changes; however a partition made at an early stage can hardly be changed later. Tree structures such as R*-tree and SR-tree have the same problem. In order to keep the space partition efficient, these tree structures force data reinsertions to roll back the previous partition. We have not developed the reinsertion algorithm for the hash structure because this problem was not recognized until the experiment result was analyzed. However, since insertions and deletions are done off-line, instead of performing reinsertions, we could rebuild the hash structure periodically.

In the experiments, we also measure the number of splits and merges happening during the insertions and deletions. Comparing the two graphs in Figure 5.9, we can see that setting the merge-threshold at 90% reduces a large number of split and merge operations without degrading space utilization too much. The results for 70% and 50% merge-thresholds are the same as for 90% because the insertions and deletions interleave so evenly that the disk block occupancy doesn't change much: actually a 90% merge-threshold already results in no merges.



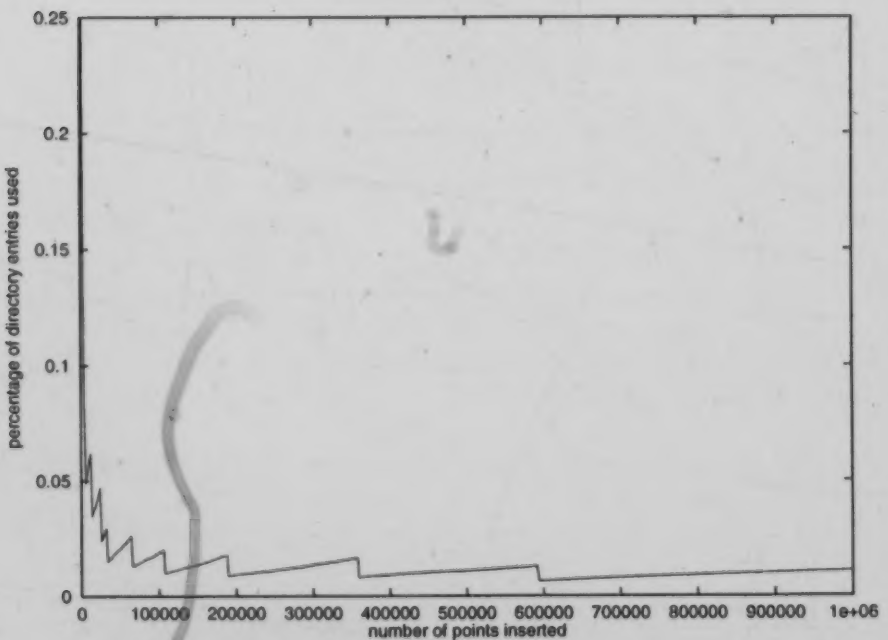
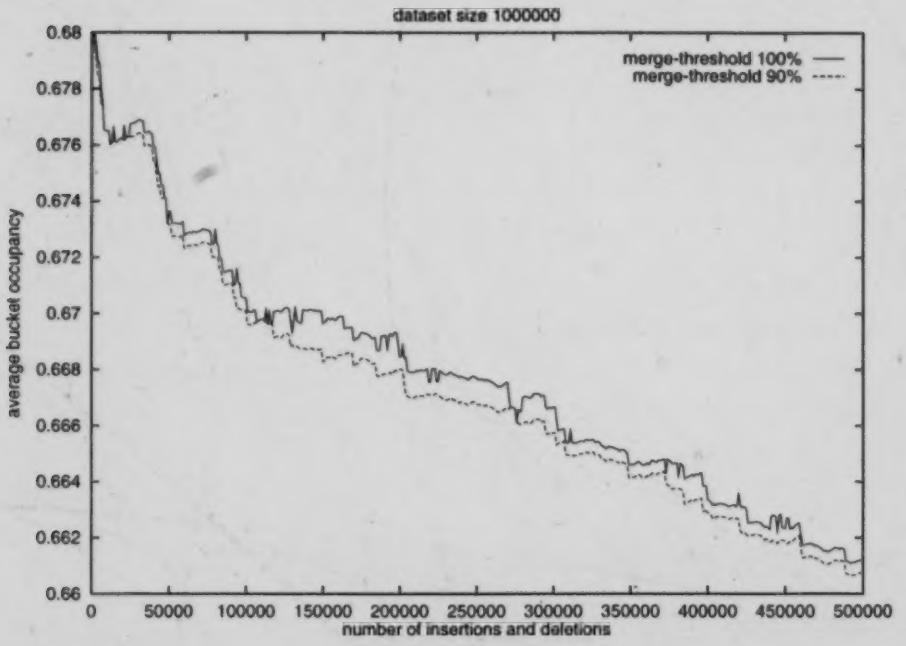


Figure 5.8: Space utilization of a continuously growing dataset



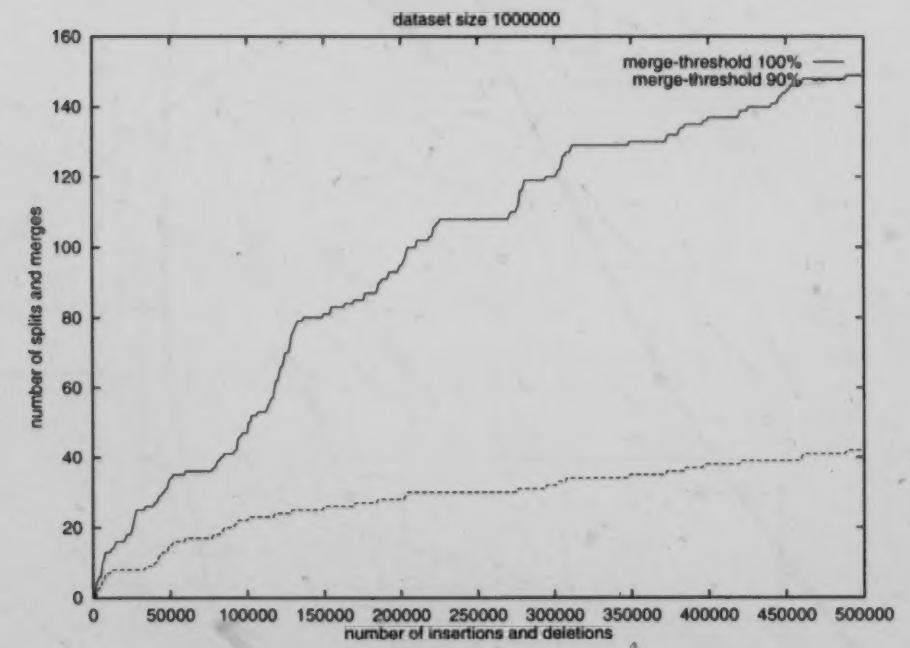


Figure 5.9: Space utilization of a steady-state dataset

Chapter 6

Colour Histogram Similarity Queries in the DISIMA System

The DISIMA system [31], developed by the Database Research Group at the University of Alberta, is extended to handle colour histogram similarity queries. The proposed three-dimensional extendible hashing is implemented and integrated into the DISIMA system. This chapter introduces the DISIMA system, particularly the extension which handles colour histogram similarity queries and the integration of the indexing structure.

6.1 The DISIMA Model

A data model is defined as a collection of mathematically well-defined concepts to express both static and dynamic properties of data intensive applications. This section presents the DISIMA model.

6.1.1 The Model Components

The DISIMA models data using an object-oriented method. The DISIMA data model aims at efficient representation of images to support a wide range of queries. Typical image queries refer both to image and the objects within images. An example query is "Find an image that contains a given object and the colour distribution of this image is similar to a sample image." The objects in images are referred to as *salient objects*. The DISIMA model, as depicted in Figure 6.1 is composed of two blocks: the image block and the salient object block. A block is defined as a group of semantically related entities.

The image block is made up of two layers: the image layer and the image representation layer. Distinguishing an image from its representation maintains an independence between them. In the image layer, the image hierarchical type classification is defined. Figure 6.2 gives an example of the type classification for news and medical image management. Various image representations may exist, such as GIF, JPEG, etc.

DISIMA views the semantic content of an image as a set of salient objects (i.e., interesting entities in the image). The salient object block is designed to handle salient object

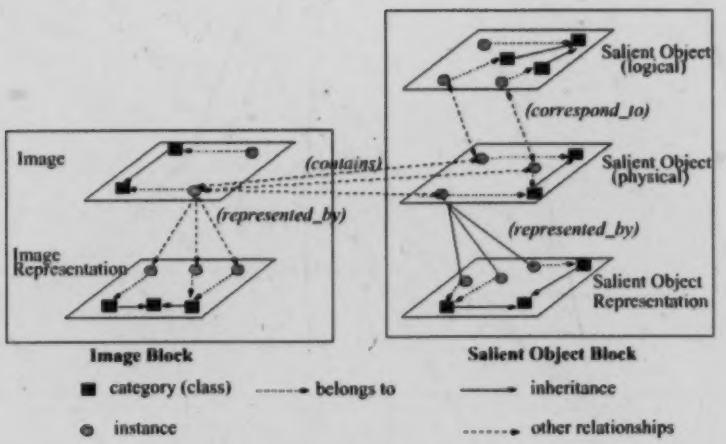


Figure 6.1: DISIMA model overview

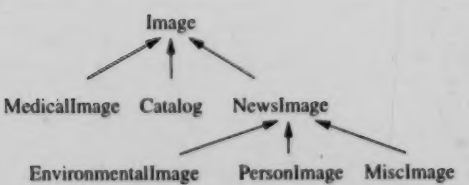


Figure 6.2: An example of image hierarchy

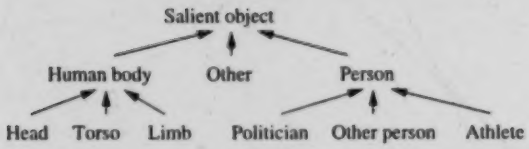


Figure 6.3: An example of salient object hierarchy

organization. DISIMA distinguishes two kinds of salient objects: physical and logical salient objects. A *logical salient object (LSO)* is an abstraction of a salient object that is known and can be defined for some application. For example, an object may be created as an instance of type *Politician* to represent Prime Minister Chrétien. The object Chrétien is created and exists even if there is yet no image in the database in which Prime Minister Chrétien appears. A *physical salient object (PSO)* refers to the physical appearance of a salient object in an image. For example, a physical salient object P_objectChrétien_1 may be created for a particular appearance of Chrétien in an image. The logical salient object maintains the generic information that might be stored about this object of interest (e.g., name, sex, spouse). The physical salient object maintains the information of the particular appearance of the object (e.g., posture, localization, shape). Salient objects may be organized in a hierarchical structure, as shown by Figure 6.3. As with the images, the representation of salient objects is separated from their content information.

6.1.2 The Type System Overview

Figure 6.4 shows a high level view of the classes used in the DISIMA type system. The Image, Image Representation, PSO, and LSO classes have been introduced before. The MBB (Minimum Bounding Box) class defines the spatial feature; the Geometric Object class defines the shape feature; the Texturegroup class defines the texture feature; the Colourgroup class and the Multi-scale Colour Histogram class define the colour feature. The straight line connecting two classes represents the relationship between the classes. (See Section 3.4 for details about multi-scale colour histograms.) The numbers on the straight lines indicate the cardinality of the relationship. For example, there is a one-to-one relationship between Image class and Multi-scale Colour Histogram class: the multi-scale colour histograms are stored in linear quadtrees (see Section 3.4). The integrated indexing structure is used to index average colours of these colour histograms to facilitate image/sub-image queries by colour histogram matching, which will be introduced in Section 6.4.

6.2 The DISIMA Architecture

The DISIMA architecture is shown in Figure 6.5. It is composed of the interfaces, the meta-data manager, the image and salient object manager, the image and spatial index manager, and the object index manager. The interfaces provide several ways (visual and alphanu-

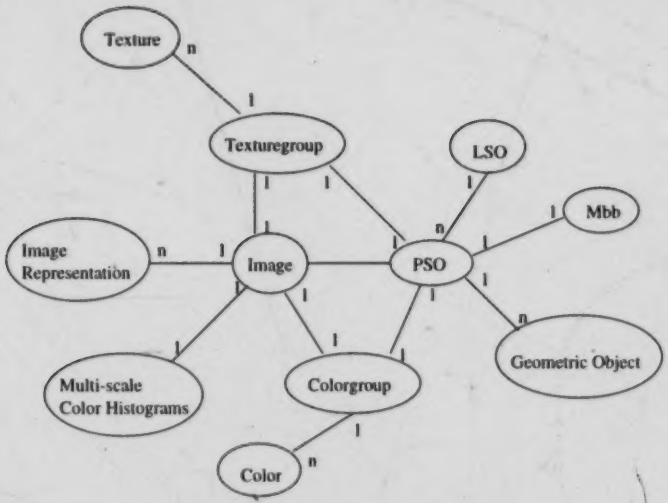


Figure 6.4: DISIMA type system overview

meric) to define and query image data. MOQL [25] (Multimedia OQL) is a textual query language that extends the standard object-oriented query language OQL with multimedia functionalities. VisualMOQL [32, 42] is based on MOQL and provides a graphical user interface for querying images. A query specified using VisualMOQL is translated into MOQL to make use of the MOQL parser and query processor. (See Section 6.3 for details about MOQL and VisualMOQL.) DISIMA is built on top of object repositories (ObjectStore for the current prototype). However, the object-oriented indices these object repositories provide (if any) may not fit with DISIMA requirements. This is why the image and spatial index manager, and object index manager, are included in the DISIMA architecture. The indexing structure that is integrated into the DISIMA system (see Section 6.4) is part of this module. The meta-data manager handles meta-information about images and salient objects. The image and salient object manager implements the DISIMA data model.

6.3 MOQL and VisualMOQL

MOQL is a multimedia extension of OQL. OQL is close to SQL [9] with object-oriented extensions. An OQL query is a function which returns an object whose type may be inferred from the operators contributing to the query expression. The basic statement of OQL is:

select [**distinct**] projection_attributes
from query [[as] identifier] {, query [[as] identifier] }
[**where** query]
[**group by** partition_attributes]
[**having** query]
[**order by** sort_criterion {, sort_criterion}]

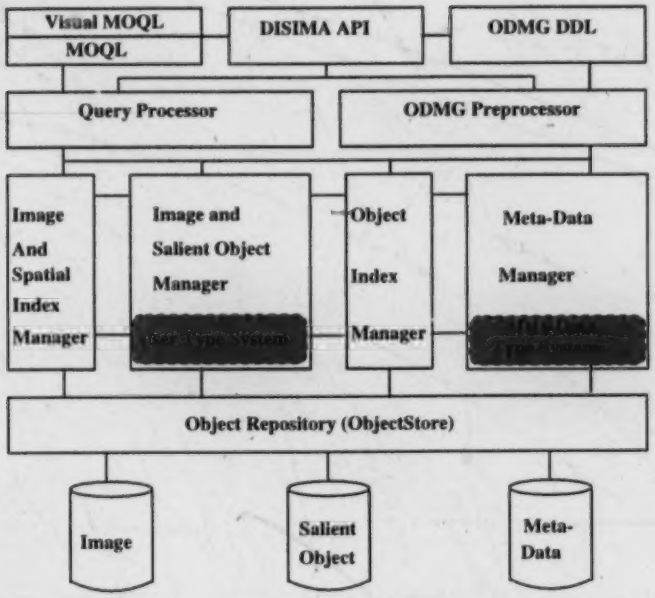


Figure 6.5: The DISIMA architecture

Most extensions introduced to OQL by MOQL are in the **where** clause, in the form of predicate expressions that deal with multimedia data. These predicate expressions include the *spatial_expression, contains_predicate, and similarity_expression.* The *spatial_expression* specifies spatial conditions using spatial objects (such as point, circle, line, etc.), spatial functions (such as length, area, etc.), and spatial predicates (cover, disjoint, etc.). The *contains_predicate* has the basic form of *image* **contains** *salient object*; it checks whether or not a salient object is in a particular image. The *similarity_expression* checks whether two images are similar with respect to some metric.

For querying images by colour histogram matching, two kinds of *similarity_expression* are used to check if two images are similar. One is whole-image similarity queries. For example:

select m
from image m
where m.colour_histogram **similar** example e1
precision 1 **similarity** 0.8

This query is to retrieve all the images that are similar to the user-provided image e1, with respect to colour histogram matching at precision level 1, with the similarity threshold 0.8. (See Section 3.5 for multi-precision similarity queries.)

The other kind of expression is querying sub-images. For example:

select m
from image m

where m.colour_histogram **similar** i.colour_histogram
quadrants (1, 2) **similarity** 0.6

This query is to retrieve all the images whose particular regions are similar to the user-provided image i, with respect to colour histogram matching with the similarity threshold 0.6. (See Section 3.6 for sub-image similarity queries and the scheme for specifying regions.) Note that the precision levels are not defined for the sub-image queries; that is, all the sub-image queries are carried out at the first precision level.

Colour histogram matching can be used with other predicate expressions. For example, the following MOQL query retrieves images that contain a person whose last name is Chrétien and the images' colour histograms are similar to the user-provided image i at precision level 2 with the similarity threshold 0.9.

select m
from image m, person p
where p.lastname='Chrétien' **and**
m.colour_histogram **similar** i.colour_histogram
precision 2 **similarity** 0.9

The above queries are all simple queries. Several simple queries can be connected using *and, or, not* operators to form a compound query. For example, the query: "Find image with 2 people next to each other without any building, or images with buildings without any people" can be expressed as

select m
from image m, building b1, person p1, person p2
where m **contains** b1 **and**
m **not in** (**select** m1
from image m1, person p3
where m1 **contains** p3) **or**
m **contains** p1 and m **contains** p2 **and**
p1.MBB **west** p2.MBB **and**
m **not in** (**select** m2
from image m2, building b2
where m2 **contains** b2)

Posting a MOQL query requires users to know the syntax of MOQL, and although users may have a clear idea of the kind of images they are interested in, the expression of the query using MOQL may not be straightforward. VisualMOQL provides an easier way to express queries, and then translates them into MOQL. VisualMOQL is a visual query interface, as shown by Figure 6.6. Users can choose the image class they want to query and the salient objects they want to see in the images. They can also specify the maximum number of images they want, and the similarity threshold. The working canvas is where

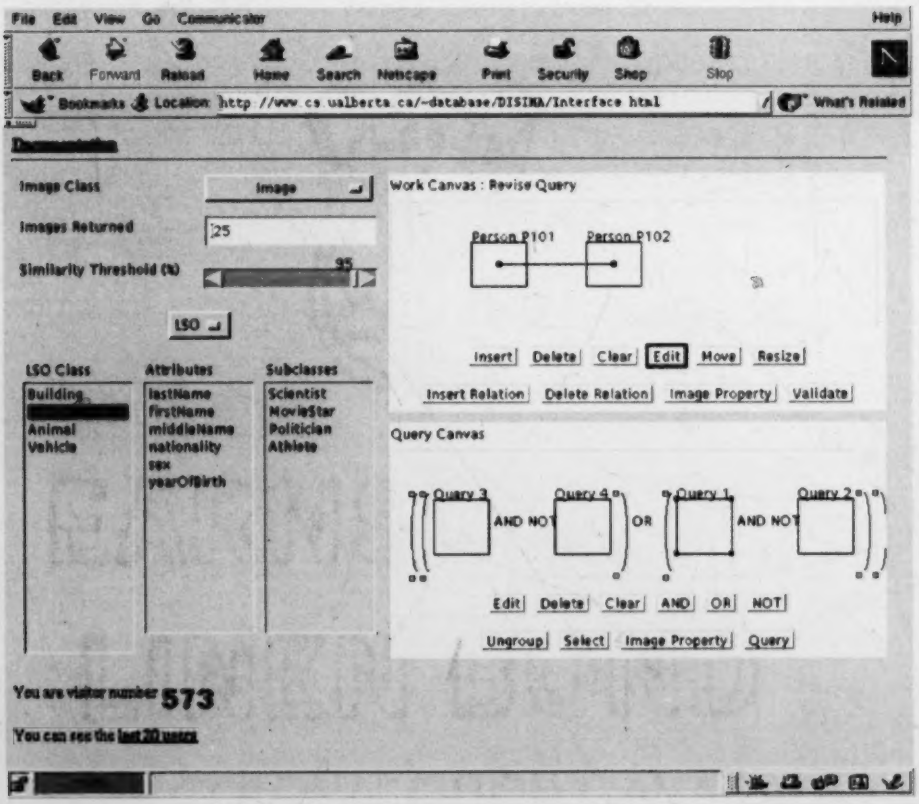


Figure 6.6: VisualMOQL interface

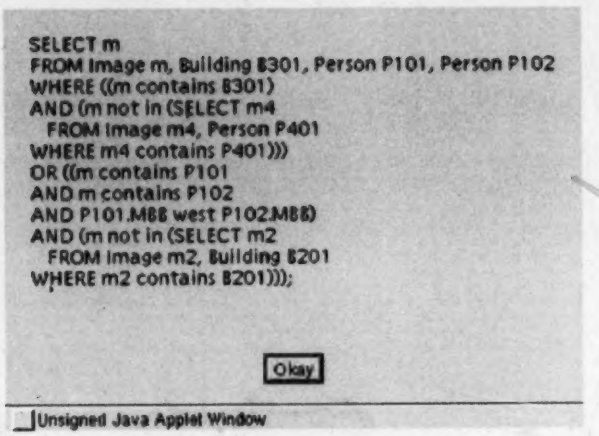

SELECT m
FROM Image m, Building B301, Person P101, Person P102
WHERE ((m contains B301)
AND (m not in (SELECT m4
FROM Image m4, Person P401
WHERE m4 contains P401)))
OR ((m contains P101
AND m contains P102
AND P101.MBB west P102.MBB)
AND (m not in (SELECT m2
FROM Image m2, Building B201
WHERE m2 contains B201))));
Okay
Unsigned Java Applet Window

Figure 6.7: Query translation

users construct simple queries. They can insert the salient objects that they want to see in images, into the working canvas. The spatial relationships may also be specified between the salient objects. For example, Figure 6.6 shows that there are two person objects in the working canvas, and a spatial relationship is defined between them: person P101 is to the west of person P102. The colour, texture, shape properties of images and salient objects can be specified through the dialog boxes provided (one of the dialog boxes is shown in Figure 6.8). After users finish constructing a simple query in the working canvas, it is moved into the query canvas. Several simple queries are combined in the query canvas to form a compound query. Finally, the user presses the query button to submit the query. The VisualMOQL query specified in the query canvas will then be translated into the MOQL query shown in Figure 6.7.

Figure 6.8 shows the dialog box for specifying image properties. This dialog box is brought up by pressing the image property button under the working canvas. The right part of the dialog box is for textual properties such as title, publisher, create date, etc. The left part of this dialog box is for colour histogram similarity matching. Users can specify the similarity threshold and provide an example image. This example image may come from the file system or the previous query results. For example, in Figure 6.8, an apartment image is provided from the file system. The lower left part of this dialog box represents the image stored in the database. For whole-image queries, users can specify the precision level by choosing the corresponding grid partitions: 1×1, 2×2, 4×4, and 8×8, and then selecting the whole image. For sub-image queries, users can select the region that they want to query on. However, the size and position of the region is limited by the grid partitions that the system provides. For example, the red rectangle in Figure 6.8 represents the query region, and this region can only be specified when the 8×8 partition is chosen.

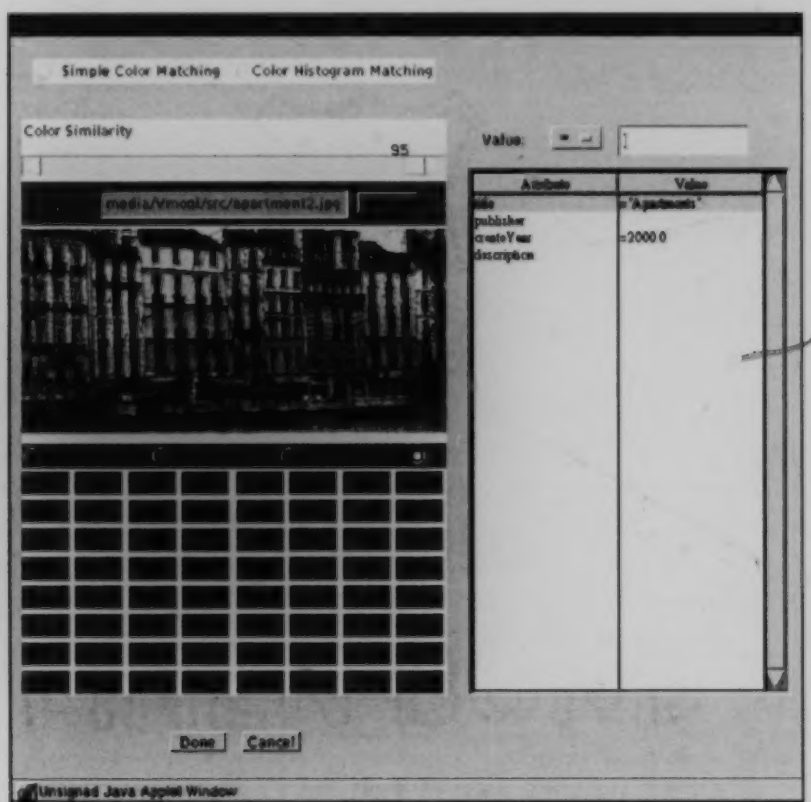


Figure 6.8: Image property dialog

6.4 Integration of Indexing Structure

The DISIMA system is extended to handle multi-precision similarity queries and sub-image similarity queries. The extension includes the following. A new class Multi-scale Colour Histogram to DISIMA type system: a Multi-scale Colour Histogram object stores the multi-scale colour histogram of an image (Section 6.1.2). MOQL language is extended to express multi-precision similarity queries and sub-image similarity queries (Section 6.3); the MOQL parser, which was written in yacc, is modified to parse the extended MOQL language. VisualMOQL is extended accordingly with an interface that allows users to issue colour histogram similarity queries (Figure 6.8). The other important extension is to build indices to facilitate colour histogram similarity searches and modify the query processor to utilize the indices.

Three-dimensional extendible hashing is designed to index average colours of the images and their quadrants. This index structure is implemented to index the images stored in the DISIMA system. If the images are unorganized in databases, then all the images will be simply inserted into one index structure. However, the images in the DISIMA system are organized as hierarchical image classes, so the index structure have to cooperate with the hierarchy of images. The index structure can be built only for the root image class, i.e, all the images are in the same index structure as if there were no image hierarchy. This method is not good for query performance. Since the image query is started by querying the database on semantic features first, i.e., by specifying image classes and salient objects, depending on the position in the image hierarchy of the class against which the query is posed, an amount of work may be wasted in searching the images that are not needed. The other choice is to build an index for each of the image classes. This method provides better query performance. However, since images in a leaf image class are in its super-classes at the same time, these images will be indexed several times. In spite of this drawback, given the general rule that space should always be traded for performance, an index per image class is a better choice. Therefore, this method is adopted to integrate index in DISIMA system. For example, given the image hierarchy shown in Figure 6.2, seven index structures are built, one for each of the image classes.

In the current implementation of DISIMA query processor, individual query conditions are processed independently, then the results from the individual query conditions are consolidated. Under this circumstance, extending the query processor to handle new types of query conditions can simply be done by adding new code without changing the previous code that handles the other types query conditions. A piece of code is added to the query processor to handle colour histogram similarity conditions. First, the hash structure corresponding to the image class of concern is located. The colour histogram similarity query is translated to average colour similarity query (see Section 3.3) and processed by searching the hashing index (see Section 4.3 for the search algorithm). Finally, the colour histogram comparisons are applied to the images that are retrieved by the average colour similarity

query in order to compute the final result image set.

An enhancement can be made to the query processor: instead of processing query conditions independently, one query condition can be processed by searching the query results of the other conditions. In this case, one has to decide which query condition should be processed first. Since indices have been built for average colours, the colour histogram similarity condition should be processed first with the help of indices, and then the set of images that is the result of colour histogram similarity condition is searched to process the rest of query conditions.

Chapter 7

Conclusions and Future Work

In this thesis, we present an indexing scheme for image colour similarity search. The images are retrieved based on colour histograms while the index is built on average colours (three-dimensional vectors), that are compact representations of colour histograms. Since we are indexing on three-dimensional data, we avoid not only the "dimensionality curse" but also the "clustering" problem [5]. We also present the multi-scale colour histograms concept, and make use of it to support multi-precision queries and sub-image queries.

The major contribution of this thesis is the new index structure — three-dimensional extendible hashing. Like the traditional extensible hash, it uses one more bit to discriminate data when the disk block overflows and needs to be split. The difference is that, instead of putting the bit to the end of the previous hash address, it puts it at the front. This makes the implementation of the hash directory more efficient because it can be expanded by simply attaching more space to the end, without changing the contents of the previous hash directory.

Three-dimensional extendible hashing is similar to grid files in that both use a directory to record the pointers to the disk blocks and, at the beginning, three-dimensional extendible hashing also imposes a grid on the universal space. The difference is that grid files use a multi-dimensional directory, and the directory can grow along any of the dimensions. Three-dimensional extensible hashing actually uses a one-dimensional directory; it uses the mask track to map three dimensions into one, which makes the directory grow slower than grid files.

We have run experiments to compare query performance of three-dimensional extensible hashing to the SR-tree. With regard to CPU time, the three-dimensional extensible hash is superior to the SR-tree. When the query range is small, the I/O performance of the hash structure is better than SR-tree, and vice versa when the query range is larger.

Although three-dimensional extendible hashing is designed to index three-dimensional colour values, nothing limits the hashing structure to three dimensions. Generally, the mask track can map multiple dimensions to one dimension. So, three-dimensional extensible hashing can easily be generalized to handle multi-dimensional data.

The crucial structure of three-dimensional extendible hashing is the mask track, which enables the traditional one-dimensional index structure to handle multi-dimensional data. The mask track can be integrated with other one-dimensional hashing to yield new multi-dimensional index structures, such as multi-dimensional linear hashing.

Further enhancements can be made for the hashing structure. Data organization with a disk block is worth consideration. Currently, the data are not ordered in the disk blocks, and so every data point has to be examined when a disk block is searched. A possible way to organize the data points is ordering them by their distances to the centroid point; the triangle inequality can then be used to reduce the number of comparisons. Since the distribution of image average colours are very skewed, the mask track and the hash directory are very sparse. Compression techniques may be employed to improve space utilization. Since the hashing structure is able to handle multi-dimensional data. It will be interesting to look into that how big the number of dimensions can be before the degradation of the hashing structure is noticed.

Bibliography

[1] MasterPhotos 50,000 premium photo collection. IMSI, 1997.

[2] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 322–331, Atlantic City, New Jersey, USA, May 1990.

[3] A. P. Berman. A new data structure for fast approximate matching. Technical Report 1994-03-02, Department of Computer Science, University of Washington, 1994.

[4] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? Technical Report TR1377, Department of Computer Science, University of Wisconsin-Madison, June 1998.

[5] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *Proceedings of the 7th International Conference on Database Theory*, pages 217–235, Jerusalem, Israel, January 1999.

[6] A. D. Bimbo. *Visual Information Retrieval.* Morgan Kaufmann Publishers, Inc., 1999.

[7] S. Brin. Near neighbor search in large metric spaces. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 574–584, Zurich, Switzerland, September 1995.

[8] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 426–435, Athens, Greece, 1997.

[9] R. A. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*, chapter 7, SQL — A Relational Database Language, pages 185–230. Addison-Wesley Publishing Company, 2nd edition, 1994.

[10] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing — a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, September 1979.

[11] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, 3(3/4), July 1994.

[12] C. Faloutsos and K. Lin. Fastmap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 163–174, San Jose, California, USA, May 1995.

[13] J. Foley, A. V. Dam, S. Feiner, and J. Hughes. *Computer graphics: principles and practice*, chapter 13, Achromatic and Colored Light, pages 563–604. Addison-Wesley Publishing Company, Inc., 2nd edition, 1990.

[14] M. J. Folk and B. Zoellick. *File Structures*. Addison-Wesley Publishing Company, Inc., 2nd edition, June 1992.

[15] M. Freeston. The BANG file: A new kind of grid file. In *Proceedings of the ACM SIGMOD 1987 Annual Conference*, pages 260–269, San Francisco, California, USA, May 1987.

[16] B. Funt and V. Cardei. Bootstrapping colour constancy. In *Human Vision and Electronic Imaging IV*, pages 421–428, May 1999.

[17] J. Miller G. Pass, R. Zabih. Comparing images using color coherence vectors. In *Proceedings of the 4th ACM International Conference on Multimedia*, pages 65–73, Boston, Massachusetts, USA, November 1996.

[18] V. Gaede and O. Günther. Multidimensional access methods. *Computing Surveys*, 30(2):170–231, 1998.

[19] R. S. Gray. Content-based image retrieval: color and edges. Technical Report PCS-TR95-252, Dartmouth College, Computer Science, March 1995. URL ftp://ftp.cs.dartmouth.edu/TR/TR95-252.ps.Z.

[20] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD 1984 Annual Meeting*, pages 47–57, Boston, Massachusetts, USA, June 1984.

[21] J. Hafner, H. S. Sawhney, W. Equitz, M. Flickner, and W. Niblack. Efficient color histogram indexing for quadratic form distance functions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(7):729–736, July 1995.

[22] K. V. R. Kanth, D. Agrawal, A. E. Abbadi, and A. K. Singh. Dimensionality reduction for similarity searching in dynamic databases. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 166–176, Seattle, Washington, USA, June 1998.

[23] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 369–380, Tucson, Arizona, USA, May 1997.

[24] K. Leung and R. T. Ng. Multiscale similarity matching for subimage queries of arbitrary size. In *Proceedings of the 4th Working Conference on Visual Database Systems*, pages 243–264, L'Aquila, Italy, May 1998.

[25] J. Z. Li, M. T. Özsu, D. Szafron, and V. Oria. MOQL: A multimedia object query language. In *Proceedings of the 3rd International Workshop on Multimedia Information Systems*, pages 19–28, Como, Italy, September 1997.

[26] Z. Li, O. R. Zaïane, and B. Yan. C-BIRD: Content-based image retrieval from image repositories using chromaticity and recognition kernel. In *Proceedings of the International Workshop on Storage and retrieval Issues in Image and Multimedia Databases*, pages 361–366, Vienna, Austria, August 1998.

[27] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proceedings of the 6th International Conference on Very Large Data Bases*, pages 212–223, Montreal, Quebec, Canada, October 1980.

[28] R. T. Ng and D. Tam. Multilevel filtering for high-dimensional image data: Why and how. *IEEE Transactions on Knowledge and Data Engineering*, 11(6), November/December 1999.

[29] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. H. Glasman, D. Petkovic, P. Yanker, C. Faloutsos, and G. Taubin. The QBIC project: Querying images by content, using color, texture, and shape. In *Storage and Retrieval for Image and Video Databases I*, volume 1908 of *SPIE Proceedings*, pages 173–187, San Jose, California, USA, 1993.

[30] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, March 1984.

[31] V. Oria, M. T. Özsu, L. Liu, X. Li, J. Z. Li, Y. Niu, and P. J. Iglinski. Modeling images for content-based queries: The DISIMA approach. In *The 2nd International Conference on Visual Information Systems*, pages 339–346, San Diego, California, USA, October 1997.

[32] V. Oria, M. T. Özsu, B. Xu, L. I. Cheng, and P.J. Iglinski. VisualMOQL: The DISIMA visual query language. In *Proceedings of the 6th IEEE International Conference on Multimedia Computing and Systems*, volume 1, pages 536–542, Florence, Italy, June 1999.

[33] A. Pentland, R. W. Picard, and S. Sclaroff. Photobook: Content-based manipulation of image databases. Technical Report 255, M.I.T. Media Laboratory Perceptual Computing, November 1993.

[34] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley Publishing Company, Inc., 1990.

[35] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th International Conference on Very Large Data Bases*, pages 507–518, Brighton, England, September 1987.

[36] J. R. Smith and S. Chang. Tools and techniques for color image retrieval. In *Storage and Retrieval for Image and Video Databases IV*, pages 426–437, San Diego/La Jolla, California, USA, 1996.

[37] J. R. Smith and S. Chang. VisualSEEK: a fully automated content-based image query system. In *Proceedings of the 4th ACM International Conference on Multimedia*, pages 87–98, Boston, Massachusetts, USA, November 1996.

[38] M. J. Swain and D. H. Ballard. Color indexing. *Internaltional Journal of Computer Vision*, 7(1):11–32, 1991.

[39] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179, November 1991.

[40] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *Proceedings of the 12th International Conference on Data Engineering*, pages 516–523, New Orleans, Louisiana, USA, 1996.

[41] G. Wyszecki and W. S. Stiles. *Color science : concepts and methods, quantitative data and formulae.* Wiley and Sons, Inc., New York, NY, 2nd edition, 1982.

[42] B. Xu. A visual query facility for disima image database management system. Master's thesis, Department of Computing Science, University of Alberta, April 2000.

[43] B. Yan. Content based search in multimedia databases. Master's thesis, School of Computing Science, Simon Fraser University, June 1997.

[44] P. Zezula, P. Ciaccia, and F. Rabitti. M-tree: A dynamic index for similarity queries in multimedia databases. Technical Report 7, HERMES ESPRIT LTR Project, 1996. URL http://www.ced.tuc.gr/hermes/.

[45] H. Zhang and D. Zhong. Scheme for visual feature-based image indexing. In *Storage and Retrieval for Image and Video Databases III*, pages 36–46, San Diego/La Jolla, California, USA, February 1995.

Appendix A

As mentioned in Section 3.5, in order to be used in multi-precision similarity queries, a colour histogram distance metric must satisfy the condition $d_L \leq d_H$. We prove here that a weighted Euclidean distance metric satisfies this condition.

The weighted Euclidean distance between two normalized colour histograms $X(x_1, x_2, \ldots, x_n)$ and $Y(y_1, y_2, \ldots, y_n)$ is $d(X,Y) = \sqrt{(\tilde{X} - \tilde{Y})^T \tilde{A} (\tilde{X} - \tilde{Y})}$ where $\tilde{A}$ is positive semidefinite (PSD) and $\tilde{X}$ and $\tilde{Y}$ are formed by the first $n-1$ elements of X and Y, respectively [21]. Define $\|X\| = \sqrt{\tilde{X}^T \tilde{A} \tilde{X}}$, then $d(X,Y) = \|X - Y\|$

LEMMA 1 $\|\lambda X\| = \lambda \|X\|$ *where* λ *is a scalar and* $\lambda \geq 0$.

PROOF. $\|\lambda X\| = \sqrt{(\lambda \tilde{X})^T \tilde{A} (\lambda \tilde{X})} = \sqrt{\lambda \tilde{X}^T \tilde{A} \lambda \tilde{X}} = \lambda \sqrt{\tilde{X}^T \tilde{A} \tilde{X}} = \lambda \|X\|$

LEMMA 2 $\|X + Y\| \leq \|X\| + \|Y\|$

PROOF. Since $\tilde{A}$ is PSD, there exists a matrix P so that

$$P^T P = E \tag{A.1}$$

where E is an identity matrix, and

$$P \tilde{A} P^{-1} = \Lambda \tag{A.2}$$

where Λ is a matrix that has $\lambda_1, \lambda_2, \ldots, \lambda_{n-1}$ along the principal diagonal and 0's elsewhere. From A.1, we have

$$P^{-1} = P^T. \tag{A.3}$$

Then from A.2 and A.3, we have

$$\tilde{A} = P^{-1} \Lambda P = P^T \Lambda P \tag{A.4}$$

Let $X'(x'_1, x'_2, \ldots, x'_{n-1}) = P\tilde{X}$. $\|X\|$ can be rephrased as

$$\begin{aligned} \|X\| &= \sqrt{\tilde{X}^T \tilde{A} \tilde{X}} \\ &= \sqrt{\tilde{X}^T P^T \Lambda P \tilde{X}} \qquad \text{(by A.4)} \\ &= \sqrt{(P\tilde{X})^T \Lambda (P\tilde{X})} \\ &= \sqrt{X'^T \Lambda X'} \\ &= \sqrt{\textstyle\sum_{i=1}^{n-1} \lambda_i x_i'^2} \end{aligned} \tag{A.5}$$

From A.5, we have

$$\begin{aligned} \|X+Y\| &= \sqrt{\textstyle\sum_{i=1}^{n-1} \lambda_i (x_i' + y_i')^2} \\ &= \sqrt{\textstyle\sum_{i=1}^{n-1} \lambda_i x_i'^2 + \sum_{i=1}^{n-1} \lambda_i x_i' y_i' + \sum_{i=1}^{n-1} \lambda_i y_i'^2} \\ &= \sqrt{\|X\|^2 + \|Y\|^2 + 2\textstyle\sum_{i=1}^{n-1} \lambda_i x_i' y_i'} \end{aligned} \tag{A.6}$$

According to Cauchy's inequality, i.e., $(\sum_{i=1}^{n} a_i b_i)^2 \leq \sum_{i=1}^{n} a_i{}^2 \cdot \sum_{i=1}^{n} b_i{}^2$, we know that

$$\begin{aligned} (\sum_{i=1}^{n-1} \lambda_i x_i' y_i')^2 &= [\sum_{i=1}^{n-1} (\sqrt{\lambda_i} x_i') \cdot (\sqrt{\lambda_i} y_i')]^2 \\ &\leq \sum_{i=1}^{n-1} (\sqrt{\lambda_i} x_i')^2 \cdot \sum_{i=1}^{n-1} (\sqrt{\lambda_i} y_i')^2 \\ &= \|X\|^2 \|Y\|^2 \end{aligned}$$

that is,

$$\sum_{i=1}^{n-1} \lambda_i x_i' y_i' \leq \|X\| \|Y\| \tag{A.7}$$

Using A.7 in A.6, we get

$$\begin{aligned} \|X+Y\| &\leq \sqrt{\|X\|^2 + \|Y\|^2 + 2\|X\|\|Y\|} \\ &= \sqrt{(\|X\| + \|Y\|)^2} \\ &= \|X\| + \|Y\| \end{aligned}$$

$\|X+Y\| \leq \|X\| + \|Y\|$ can be generalized to $\|\sum X_i\| \leq \sum \|X_i\|$. Now we are ready to prove $d_L \leq d_H$. We only need to prove that this inequality holds between the first precision level and the second. Then the general case can easily be proved by induction.

Let X be a colour histogram computed from an entire image, and let X_1, X_2, X_3, and X_4 be colour histograms of quadrants of the image, respectively. Let Y be a colour histogram for another image, and Y_1, Y_2, Y_3, and Y_4 be colour histograms for quadrants of the image, respectively. $d_L \leq d_H$ is equivalent to $d(X,Y) \leq \frac{1}{4}\sum_{i=1}^{4} d(X_i, Y_i)$, i.e., $\|X-Y\| \leq \frac{1}{4}\sum_{i=1}^{4} \|X_i - Y_i\|$.

PROOF. Notice that $X = \frac{1}{4}(X_1 + X_2 + X_3 + X_4)$ and $Y = \frac{1}{4}(Y_1 + Y_2 + Y_3 + Y_4)$, so

$$\begin{aligned} \|X-Y\| &= \|\frac{1}{4}\sum_{i=1}^{4} X_i - \frac{1}{4}\sum_{i=1}^{4} Y_i\| \\ &= \|\frac{1}{4}\sum_{i=1}^{4} (X_i - Y_i)\| \\ &= \frac{1}{4}\|\sum_{i=1}^{4} (X_i - Y_i)\| \qquad \text{(by LEMMA 1)} \\ &\leq \frac{1}{4}\sum_{i=1}^{4} \|X_i - Y_i\| \qquad \text{(by LEMMA 2)} \end{aligned}$$